University of California
Santa Barbara

Bridging the Theory-Practice Gap of Laplacian Linear Solvers

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Kevin Deweese

Committee in charge:

Professor John R. Gilbert, Chair
Doctor Erik G. Boman
Professor Shivkumar Chandrasekaran
Professor Xifeng Yan

June 2018

The Dissertation of Kevin Deweese is approved.

Doctor Erik G. Boman

Professor Shivkumar Chandrasekaran

Professor Xifeng Yan

Professor John R. Gilbert, Committee Chair

April 2018

Bridging the Theory-Practice Gap of Laplacian Linear Solvers

Kevin Deweese

To the science fiction writers

"Arrakis teaches the attitude of the knife - chopping off what's incomplete and saying: 'Now, it's complete because it's ended here.'" —Frank Herbert (Dune)

Acknowledgements

To my committee members Erik Boman, Shivkumar Chandrasekaran, Xifeng Yan, and especially my committee chair John Gilbert, thank you for your support and guidance.

To my other colleagues, Karen Devine, Tristan Konolige, Rich Lehoucq, Adam Lugowski, Gary Miller, Ojas Parekh, Siva Rajamanickam, Richard Peng, Veronika Strnadova-Neeley, Sivan Toledo, Haoran Xu, and Shen Chen Xu, thank you for your helpful advice and support.

To all of my amazing teachers, who are too numerous to name, thank you for your patience and dedication.

To my housemates Zach, Jessica, Viva, Yuliy, Tie, Brian, Erik, Lukas, Alysha, Wenjun, and Michael, thank you for your food and conversation, and I forgive all dishes left undone.

To my friends, both near and far, Eric, Bennett, Eric, Mary, Kyu, Aryeh, Richard, Nick, Evan, Cam, Andy, Nancy, Nick, John, Adam, Veronika, Jamie, Ben, Arya, Rone, Michael, Christina, Tristan, Collin, Sean, Courtney, Mathieu, Katie, Mehmet, Barclay, and Ryan, thank you for keeping me sane for the past seven years.

To Mom, Dad, Kyle, and Kelsey, thank you for your love and support. Special thanks to Ed for the Jocko's.

To my wife Ying-Jung, although I met you late in grad school, your love and support (and trips to the ER) helped me get to the finish line.

Curriculum Vitæ

Kevin Deweese

Education

2018	Ph.D. in Computer Science, University of California, Santa Barbara.
2011	B.S. in Computer Science, Physics, Vanderbilt University.

Publications

K. Deweese and J. R. Gilbert, *Evolution of Difficult Graphs for Laplacian Solvers* (accepted SIAM CSC 2018, awaiting publication)

K. Deweese, J. R. Gilbert, G. Miller, R. Peng, H. R. Xu, and S. C. Xu, *An empirical study of cycle toggling based Laplacian solvers*, SIAM CSC Workshop (presented), 2016, Albuquerque, NM.

E. G. Boman, K. Deweese, J. R. Gilbert, *An empirical comparison of graph Laplacian solvers*, SIAM ALENEX Workshop (presented), 2016, Arlington, VA.

E. G. Boman, K. Deweese, and J. R. Gilbert, *Evaluating the dual randomized Kaczmarz Laplacian linear solver*, Informatica 40, 2016.

R. W. Techentin, B. K. Gilbert, A. Lugowski, K. Deweese, J. R. Gilbert, E. Dull, M. Hinchey, and S. P. Reinhardt, *Implementing iterative algorithms with SPARQL*, EDBT/ICDT Workshops, 2014.

K. Deweese, J. R. Gilbert, A. Lugowski, and S. Reinhardt, *Graph clustering in SPARQL*, SIAM Workshop on Network Science, 2013.

E. G. Boman and K. Deweese, *A comparison of preconditioners for solving linear systems arising from graph Laplacians*, CSRI Summer Proceedings, 2014.

Abstract

Bridging the Theory-Practice Gap of Laplacian Linear Solvers

by

Kevin Deweese

Solving Laplacian linear systems is an important task in a variety of practical and theoretical applications. Laplacians of structured graphs, such as two and three dimensional meshes, have long been important in finite element analysis and image processing. More recently, solving linear systems on the Laplacians of large graphs without mesh-like structure has emerged as an important computational task in network analysis. A number of theoretical solvers with good asymptotic complexity have been proposed over the past couple decades, but these ideas have not made their way into practical solvers. Nor is it clear that a class of challenging problems exist which would benefit from asymptotically fast solvers. Yet it seems that one of the following should be true: either existing solvers have tighter Big-O bounds than currently believed, or there are some problems where recent asymptotically fast (but theoretical) algorithms should be useful.

This work considers the latter possibility; we aim to bridge the gap between theoretical and practical Laplacian algorithms by experimenting with Laplacian solvers and by searching for difficult test problems. We examine the performance of existing algorithms for solving Laplacian linear systems and identify the strengths and weaknesses of different methods on different test problems. We perform an extensive evaluation of the KOSZ solver, one of the recently proposed $\widetilde{O}(m)$ Laplacian algorithms. We test various extensions of KOSZ which we propose to try and improve its performance in practice. We introduce heavy path graphs, a novel class of graphs for experimenting with Laplacian solvers.

To challenge existing solver implementations, we propose the use of genetic algorithms

to create difficult test graphs for existing solvers. At the same time, these algorithms could be used to find graphs with good performance for recently proposed solvers. Searching for graphs which satisfy both objectives could be instrumental towards bridging the theory-practice gap of Laplacian solvers. We demonstrate the successful evolution of graphs which are difficult for conjugate gradient with diagonal scaling, while relatively simple for KOSZ. Such graph evolution techniques could be useful for finding graphs with a variety of combinatorial properties.

Contents

Chapter 1

Introduction

The Laplacian matrix of an undirected graph G with positive edge weights w_e is defined as $\mathcal{L}_G = \mathcal{D}_G - \mathcal{A}_G$, where $\mathcal{D}_G$ is the diagonal matrix containing the sum of incident edge weights and $\mathcal{A}_G$ is the weighted adjacency matrix. If G contains n vertices and m edges, then $\mathcal{L}_G \in \mathbb{R}^{n \times n}$ and contains $2m$ nonzero off-diagonal entries. Hereafter we omit the subscript G where it does not cause ambiguity. If $w_{i,j}$ is the weight of an edge $e_{i,j} \in G$ between vertices i and j, then the entries of the Laplacian are defined as

$$\mathcal{L}_{i,j} = \begin{cases} -w_{i,j} & i \neq j, e_{i,j} \in G \\ \sum_{e_{i,k} \in G} w_{i,k} & i = j \\ 0 & otherwise. \end{cases}$$

$\mathcal{L}$ is a symmetric, positive semidefinite, diagonally dominant M-matrix, with a nullspace containing the constant vector.

This dissertation focuses on finding solutions to linear systems of equations of the form $\mathcal{L}x = b$, where $\mathcal{L}$ and b are known and fixed, and we desire high quality approximations to x. While solving such systems is simpler than more general classes of matrices (non-symmetric, negative eigenvalues, complex valued, etc.), it remains an important task in

a variety of practical and theoretical applications. Interest in solving Laplacian linear systems also stems from interest in solving the slightly more general class of symmetric diagonally dominant (SDD) matrices. A matrix A is SDD if $A = A^T$ and $a_{i,i} \geq \sum_{i \neq j} |a_{i,j}|$ for all i. One way of solving an SDD system is to first reduce it to a Laplacian matrix [1, 2] and then solve it with a Laplacian solver. Some of the Laplacian linear systems in the following applications derive from SDD linear systems. Since the reduction from an SDD matrix to a graph Laplacian is asymptotically cheap, the theoretically best SDD solvers rely on Laplacian solvers.

1.1 Applications and Recent Algorithm Development

The earliest applications of Laplacian matrices involve the modeling of physical systems, which often require the solution of elliptic partial differential equations. Graph Laplacians, also known as discrete Laplacians, take their name from the continuous Laplace operator found in differential equations. Finite element analysis uses discrete Laplacians to find numerical solutions to the Laplace operator. In this way, solving Laplacian linear systems is an important component in studying electrical and thermal conductivity, as well as fluid flow [3]. The discretization of these physical systems produces an underlying graph known as a *mesh*, which typically has regular degree vertices of low degree as well as a 2D or 3D embedding representing the physical system.

Another traditional application of graph Laplacians is the modeling of electrical resistive networks, where the graph Laplacian is used to represent the underlying circuit diagram [2, 4]. While this is an important real-word application, the electrical network interpretation is also useful for explaining the physical meaning of Laplacian matrices to those with a more combinatorial background. These graphs are very different from the meshes found in finite element applications, with more degree variance and more complex

spatial layouts.

There are several more recent applications of the Laplacian matrix in the realm of data science. There is less physical meaning attached to these linear systems, but their solutions are still useful. Laplacian solvers can be used in various image processing tasks, such as image segmentation, inpainting, regression and classification [5]. Researchers often cite social network analysis and computational biology as potential applications of Laplacian solvers [6]. While there are a few direct uses of Laplacian solvers in this area, such as computing Katz centrality scores [7], it is important to note that in most cases the connection is indirect. These applications mostly involve community detection algorithms which rely on spectral graph information [8, 9]. Solving Laplacian linear systems in these domains is seen as stepping stone towards better eigensolvers [10]. Typically the underlying graphs in these applications (excluding image processing) have properties very different from either meshes or resistive networks, and often lack appealing visual representations.

Practical solvers exist for many of the above applications, though none of them are believed to scale close to linearly in the number of edges m. In the last few decades, the theoretical computer science community has done significant work on asymptotically efficient Laplacian algorithms. Spielman and Teng [11] showed how to use spectral sparsifiers to solve these problems with linear (in m) times polylogarithmic work, an $O(m \log^c n)$ algorithm with a potentially large constant c. Other researchers built upon this result to develop algorithms with simpler descriptions and better Big-O bounds [6, 12, 13]. While some researchers refer to this collection of $O(m \log^c n)$ algorithms as *nearly linear*, we prefer the term *asymptotically fast*, or simply $\widetilde{O}(m)$.

The development of asymptotically fast Laplacian algorithms sparked the *Laplacian Paradigm* [14], the use of $\widetilde{O}(m)$ Laplacian solvers as a primitive operation in a variety of theoretical applications. These include a number of fundamental graph problems

(maximum and multi-commodity flow [15, 16], graph sparsification [17], generation of random spanning trees [18]), machine learning tasks (document classification [19]), network analysis (spectral clustering without eigen-decomposition [20]), and computing fundamental matrix properties (approximate Fiedler eigenpair [14], matrix exponential [21]). We compiled this list with help from a variety of useful references [4, 6, 14]. Note that in most cases the current state of the art methods for these problems do not rely on Laplacian solvers. However, if we had fast, practical Laplacian solvers based on $\widetilde{O}(m)$ algorithms, we could potentially solve these problems much faster than what is currently known.

1.2 The Theory-Practice Gap: Dissertation Outline

Despite a strong incentive to develop practical $\widetilde{O}(m)$ solvers, these theoretical ideas have yet to displace traditional solver techniques used for decades. Furthermore, it is unclear on which types of problem graphs $\widetilde{O}(m)$ algorithms could offer improvement over existing techniques. These unknowns compose the theory-practice gap referred to in the title of this work. As Sivan Toledo noted [22], we should either be able to prove tighter bounds on solvers currently used in practice, or there must be some class of problems where $\widetilde{O}(m)$ algorithms outperform existing solvers. Since many great minds have already spent several decades pursuing tighter bounds on existing solvers, we aim to explore the second possibility.

At the onset of this dissertation work, the theory-practice gap was quite large, with little communication between theoretical computer scientists developing $\widetilde{O}(m)$ algorithms and high performance computing experts developing practical solver libraries. This has changed in the past few years, and a few promising contenders for practical $\widetilde{O}(m)$ solvers have emerged as a result [23, 24]. In this body of work, we present our own efforts and findings towards bridging the theory-practice gap of Laplacian solvers, which we outline

below.

We discuss existing and theoretical algorithms in Chapter 2. To have an understanding of the current state of the art solvers used in practice, and to have a baseline for comparing future solvers, we performed an empirical evaluation of existing solvers and present our findings in Chapter 3. As the KOSZ solver of Kelner et al. was one of the first $\widetilde{O}(m)$ algorithms with a simple description, we implemented it and provide extensive empirical results in Chapter 4. We also include in Chapter 4 a discussion of heavy path graphs, an interesting graph model we developed for our experiments. In order to find interesting test problems that challenge existing Laplacian solvers, we developed a genetic algorithm for graph evolution, which we present in Chapter 5.

1.3 Related Problems

1.3.1 Normalized Laplacians

The *normalized Laplacian* of a graph is the matrix $\mathcal{N} = \mathcal{D}^{-1/2}\mathcal{L}\mathcal{D}^{-1/2}$, whose rows and columns are scaled symmetrically to place ones on the diagonal. Like $\mathcal{L}$, the matrix $\mathcal{N}$ is symmetric and positive semidefinite, with a one-dimensional nullspace if the graph is connected. Some applications use the normalized Laplacian, and some authors [25] define "Laplacian" to mean $\mathcal{N}$, and call $\mathcal{L}$ the *combinatorial Laplacian*.

Formally, solving linear systems on $\mathcal{L}$ and on $\mathcal{N}$ are equivalent, since $\mathcal{L}x = b$ is equivalent to

$$\mathcal{N}(\mathcal{D}^{1/2}x) = (\mathcal{D}^{-1/2}\mathcal{L}\mathcal{D}^{-1/2})(\mathcal{D}^{1/2}x) = \mathcal{D}^{-1/2}b.$$

However $\mathcal{L}$ and $\mathcal{N}$ have different eigenvalues, so iterative methods may converge differently on them. Indeed, the normalized linear system is exactly the unnormalized linear system with a Jacobi preconditioner. Typically $\mathcal{N}$ is better conditioned than $\mathcal{L}$.

It is difficult to compare solver performance between $\mathcal{L}$ and $\mathcal{N}$. When solving both systems to the same residual error tolerance, the actual error of the normalized system is typically less because it is better conditioned. Our primary focus in this paper is on methods for the unnormalized Laplacian, including preconditioned methods. We do not include normalized Laplacian results, but it is an interesting related problem.

1.3.2 Spectral Graph Theory and Eigensolvers

Understanding the relationship between the properties of a graph G and the eigenvalues of $\mathcal{A}_G$, $\mathcal{L}_G$, and $\mathcal{N}_G$ is a topic known as spectral graph theory, a fundamental intersection between graph theory and linear algebra. A course or text on spectral graph theory [25, 26] is very useful to Laplacian solver researchers as we lean on this field to provide insight into graph properties and solver behavior.

Developing eigensolvers for finding graph spectra is a separate research direction. As mentioned previously, many applications of $\mathcal{L}$ are more directly related to finding the eigenpairs (λ, v) of the system $\mathcal{L}v = \lambda v$. Less is known about fast Laplacian eigensolvers for finding arbitrary parts of the spectrum, though it is certainly an important problem of interest [27, 28, 29].

Chapter 2

Solver Overview

This chapter provides details of a variety of different Laplacians solvers, which we broadly separate into two categories. The first category is what we call traditional solvers, which perform well in practice and have been optimized by the high performance computing community for decades. However, these methods are not believed to be $\widetilde{O}(m)$ for arbitrary graphs. The second category is theoretical solvers with provably good asymptotics. These solvers generally lack practical implementations (though this is beginning to change). Nevertheless, they are of great interest as their $\widetilde{O}(m)$ bounds indicate rich potential if practical implementations became available. In between these two categories we will briefly discuss support-graph preconditioners. While these methods were neither traditionally used in practice, nor have provably good asymptotics, they are an important bridge between these categories, both historically and in the context of this dissertation. We will discuss them further in Chapter 3.

2.1 Traditional Solvers

Algorithms for solving sparse linear systems come in two varieties, direct and iterative methods [30]. Direct methods attempt to solve the problem in a finite number of operations, and in the absence of rounding errors, will provide exact solutions. For Laplacian matrices,

which are positive semidefinite, the natural direct method is a Cholesky decomposition [31], which factors the matrix into symmetric lower and upper triangular matrices. Reordering is performed before factorization to reduce fill in the factors. Forward and backwards substitution are then performed to solve the linear system.

Direct methods can be prohibitively expensive both in terms of time and memory use, so iterative methods are often used instead. Iterative methods form a sequence of improving approximations to the solution, which terminates when the approximation meets some convergence criteria. Two of the most popular iterative methods for solving linear systems are multigrid solvers and Krylov subspace solvers.

Multigrid solvers [32] construct a recursive hierarchy of coarse approximations to the original matrix $A = A^1 \simeq A^2 \simeq A^3 ... \simeq A^k$, where A^{i+1} should have fewer variables and be easier to solve than A^i. The solution at each level x^i is approximated by the solution at x^{i+1}. Often a direct method is used to solve for x^k on the coarsest level A^k exactly. Special operations called *prolongation* and *restriction* are used to transfer solutions between the levels of the hierarchy. Approximating the solution on a coarser level reduces the smooth, or low frequency error. To remove high frequency error (leaving only smooth error), a *smoother*, such as Jacobi or Gauss-Seidel, is applied on the finer levels. By repeatedly restricting and interpolating solutions up and down the hierarchy, multigrid can often find solutions quickly and reliably. To improve robustness, they are often used inside a Krylov solver which we explain below.

Krylov subspace methods form a subspace consisting of an initial vector times successive powers of the problem matrix

$$K_k(A, b) = \text{span}\{b, Ab, ..., A^{k-1}b\},$$

where b is the right hand side of the linear system to be solved. The methods search for a

solution within this subspace. For Laplacian matrices, which are symmetric and positive semidefinite, the typical Krylov method is the conjugate gradient method [30].

The conjugate gradient algorithm's complexity can be theoretically bounded in terms of the square root of the spectral condition number, which is the ratio of largest to smallest nonzero eigenvalue of the problem matrix.

$$\kappa(A) = \frac{\lambda_{\max}(A)}{\lambda_{\min}(A)}.$$

Typically a transformation called a preconditioner is applied to the linear system to improve the convergence behavior of iterative methods. Formally, a preconditioner is a matrix M that approximates A (that may or may not be formed explicitly), and is used to transform the linear system $Ax = b$ to

$$(M^{-1/2}AM^{-1/2})(M^{1/2}x) = M^{-1/2}b.$$

The preconditioned conjugate gradient algorithm's complexity is then theoretically bounded in terms of $\kappa(M^{-1/2}AM^{-1/2})$. Finding a good preconditioner is a balance between being cheap to compute and apply, and effectively improving convergence behavior. A preconditioner does not always need to be constructed explicitly, as this could be prohibitively expensive. Furthermore a preconditioner can itself be approximated by a preconditioner. This is known as multi-level preconditioning [33]. The hierarchy of matrix approximations in multigrid methods can be used as a multi-level preconditioner. The following subsections summarize common single-level and multi-level preconditioners used in practice for Laplacian solvers.

2.1.1 Preconditioned Conjugate Gradient (PCG) with Single-level Preconditioner M.

- Jacobi: This is a simple relaxation preconditioner which sets $M = D$, where D is the diagonal of the original matrix [30]. There is little setup cost as M is easily determined from the original matrix. Every solve applies M^{-1} by using knowledge of the matrix. Note that Jacobi PCG on a Laplacian is equivalent to unpreconditioned CG on the normalized Laplacian.

- Symmetric Gauss-Seidel (SGS): This is a relaxation preconditioner which sets $M = (D + L)D^{-1}(D + U)$ where D, L, and U are the diagonal, lower triangular, and upper triangular parts of the decomposition $A = L + D + U$ [30]. There is little setup cost as M is not explicitly formed. Every solve applies M^{-1} by using knowledge of the matrix.

- Incomplete Cholesky Factorization (IC): This sets $M = LDL^T$ where the setup cost is approximately factoring A into $A \simeq LDL^T$ by dropping entries during the factorization [30]. M is not explicitly formed; instead, a solve phase applies forward and backward substitution with the factors.

2.1.2 Preconditioned Conjugate Gradient (PCG) with Multi-level Preconditioner.

- Algebraic Multigrid: Used inside PCG, M is not explicitly formed but is instead the linear operator equivalent to applying a single iteration of a multigrid solver.

  There are many application specific multigrid solvers that rely upon the geometrical structure of the problem matrix [34]. Of greater interest to this work is algebraic multigrid (AMG), an umbrella term for multigrid methods that solve on general

matrices with no structural information [35]. Most AMG codes, including those found in Trilinos and PETSc, are challenged by graphs with an irregular degree distribution. To address this problem, Livne and Brandt proposed the Lean Algebraic Multigrid [10] and Napov and Notay proposed a similar multigrid solver [36, 37].

2.2 Support-Graph Preconditioners

Support-graph preconditioning was introduced by Vaidya in the early 1990's [38]. The original work remains unpublished but good explanations and improvements of the original ideas exist due to several researchers interested in Vaidya's work [1, 3, 39, 40]. A support graph is a sparse approximation of the original target graph, which is used as a single level preconditioner in Krylov subspace methods. The simplest support graph is simply a spanning tree of the original graph. To improve the approximation of the tree to the original graph, and thus improve the condition number of the preconditioned system, a very small number of additional non-tree edges can also be added back to the tree. The resulting sparse preconditioner can be completely factored much faster and with less memory usage than the original graph. Whereas incomplete Cholesky preconditioners rely on an incomplete factorization of a complete matrix, support-graph preconditioners use a complete factorization of an incomplete matrix.

The best spanning trees for approximating the original graph are known as *low-stretch spanning trees* [41]. Stretch is a notion of the penalty you must pay to travel between two vertices, via spanning tree traversal, instead of being able to use the original off-tree edges. They are interesting and important combinatorial objects which have been refined since the first nontrivial construction given by Alon et al. [42, 43, 44, 45]. However, these ideas are mostly theoretical; there is a dearth of low-stretch tree generation software. Instead one can use a maximum weight spanning tree as a simple low-stretch spanning tree. The

first support graph preconditioners relied on maximum weight spanning trees.

Constructing a preconditioner from a spanning tree alone bounds the condition number by $O(nm)$, which used inside PCG yields a solver bounded by $O(m\sqrt{nm})$ for general graphs. There is then a tradeoff between adding additional edges to improve the condition number, at the cost of a more expensive factorization. The theoretically optimal number of edges to add is $n^{1/4}$, which yields a total solution time (including finding the edges, factoring the preconditioner, and PCG iterations) of $O(n^{1.75})$, for sparse graphs with $m = O(n)$. For planar graphs this is improved to $O(n^{1.2})$.

There is some experimental work in this area. Chen and Toledo implemented Vaidya's preconditioners and compared them against a variety of incomplete Cholesky preconditioners on 2D and 3D meshes [46]. They found Vaidya's preconditioners to have near constant convergence rate on a variety of problems, concluding that they could be a good choice for difficult problems for which incomplete Cholesky stagnates. However, they found the eigenvalues of the Vaidya preconditioned systems to be spread out, indicating a tight worst case Chebyshev polynomial approximation bound. This indicates poor performance for Vaidya's preconditioners in general, as the number of required iterations tends to be close to the worst case bounds. They verified this experimentally by showing problems in which Vaidya performed very poorly compared to incomplete Cholesky.

A similarly named, but fundamentally different technique called support tree conjugate gradient (STCG) was proposed by Gremban and Miller [2]. This support tree does not refer to a spanning tree of the original graph. Instead it refers to a hierarchical tree formed by recursively coarsening the original graph, where the leaves of the tree are the original graph's vertices, and new vertices are added during the coarsening. This process appears similar to a multigrid method, but all levels of the hierarchy exist in a single graph. This creates a linear system with larger dimension than the original matrix and similar sparsity. The tree is then used as a preconditioner inside CG to create an $O(m\sqrt{n \log n})$ solver.

This dissertation includes our own experimentation with support-graph ideas. We implemented spanning-tree preconditioners in the Trilinos software package [47, 48] and reported serial and parallel performance results [49, 50]. Unlike Chen and Toledo, we looked at test problems from network science domains. Similarly we found these preconditioners to have advantages. However, they are inferior to existing methods on most problems. We also showed that these techniques scale well in distributed memory. We include our spanning-tree preconditioner implementation in the suite of solvers we compare in Chapter 3. While support-graph preconditioners have not found their way into practice, the underlying combinatorial analysis contributed to the further development of fast Laplacian solvers described in the next section.

2.3 Solvers With Provably Good Asymptotics

This section provides brief descriptions of some of the notable algorithms and ideas used in $\widetilde{O}(m)$ solvers. One way to present this would be to provide a chronological list of the best known $\widetilde{O}(m)$ bounds. Instead we divide this section categorically, as separate ideas have developed independently. We do not claim that this is an exhaustive list of fast Laplacian solvers. Nor do we discuss the extensive body of related work, other than to say that researchers have worked to apply these methods to more general matrices than Laplacians [24], to prove meaningful statements about the numerical accuracy of these methods [51], and to develop parallel extensions of these methods [52].

2.3.1 Spectral Sparsifiers

The first $\widetilde{O}(m)$ solver, introduced by Spielman and Teng [11, 53] in 2004, relied upon spectral sparsifiers, which are very sophisticated subgraph preconditioners. These sparsifiers, which Spielman and Teng called ultra-sparsifiers, have tight bounds on the

number of off-diagonal matrix entries, while remaining good spectral approximations to the original graph. More formally, they defined a (k, h) ultra-sparsifier S of the original matrix A as a matrix where $S \preccurlyeq A \preccurlyeq kS$ ($A \preccurlyeq B$ indicates $B - A$ is positive semi-definite), $nnz(S) \leq 2(n-1) + 2hm/k$, and $S_{i,j} \neq 0$ only if $A_{i,j} \neq 0$. By recursively constructing a hierarchy of these ultra-sparsifiers, and using them inside a Krylov solver, they described an $O(m \log^c n \log(1/\epsilon))$ solver, with a potentially large constant c which they did not attempt to bound. To our knowledge nobody has tried to implement this method due to its complicated description. Still this was an important achievement in the theoretical computer science community.

Koutis, Miller, and Peng [12, 54] described improved spectral sparsifiers which they called incremental sparsifiers, but which are now sometimes referred to as augmented trees. They used these sparsifiers to produce a Laplacian algorithm with a bound of $O(m \log n \log \log n \log(1/\epsilon))$. These sparsifiers were developed to reduce the setup time and recursive depth required of the Spielman-Teng sparsifiers, while remaining good spectral approximations. Generating these sparsifiers starts with a low-stretch tree. The algorithm then scales the edge weights of this tree by a constant factor and then samples off-tree edges with probability proportional to the stretch of the scaled tree. Actually, the off-tree edges are split into parallel edges, each owning some fraction of the original edge weight. These sampled fractional edges are added to the scaled tree to form the sparsifier. One benefit in terms of setup cost is that the low-stretch tree can be used at all levels of the recursive hierarchy.

A more recent spectral sparsifier technique proposed by Feng [23] starts with a low-stretch spanning tree and adds edges back to the tree based on spectral perturbation analysis. The author did not include a Big-O bound of a Laplacian algorithm relying on these sparsifiers, but did claim to produce a σ-similar sparsifier S (defined as S such that $S/\sigma \preccurlyeq \mathcal{L} \preccurlyeq \sigma S$) with $n - 1 + O(\frac{m \log n \log \log n}{\sigma^2})$ edges, whose construction is

$O(m)$. Furthermore, the author implemented this technique and presented promising experimental results on several electrical network problems.

2.3.2 Dual Randomized Kaczmarz and Accelerated Coordinate Descent

In 2013, Kelner et al. [6] proposed the first $\widetilde{O}(m)$ Laplacian solver with a simple algorithmic description. The authors originally viewed this combinatorial algorithm through the lens of electrical network theory. At a high level the Kelner et al. method randomly selects cycles in a resistive network, and updates electrical flows around these cycles to minimize electrical power in the network, while satisfying Kirchoff's voltage law. The resulting algorithm has work bounded by $O(m \log^2 n \log\log n \log(1/\epsilon))$. We present the electrical network interpretation in more detail in Chapter 4, alongside our experimental work on this algorithm.

Another interpretation of the Kelner et al. algorithm, which turns out to be more useful for later improvements upon this idea, is to view the cycle updates as randomized Kaczmarz projections [55, 56, 57] in the dual space of the graph. Lee and Sidford [51] combined the randomized Kaczmarz projection ideas with the accelerated coordinate descent ideas of Nesterov [58] to produce an $O(m \log^{3/2} n \sqrt{\log\log n} \log(\log(n)/\epsilon))$ Laplacian solver.

2.3.3 Approximate Gaussian Elimination

Recently in 2016, Kyng and Sachdeva [13] proposed a purely linear algebraic algorithm that does not rely on any underlying graph objects such as low-stretch spanning trees or sparsifiers. This is an exciting development from both an implementation standpoint, and for generalization to non-Laplacian matrices. At a high level this method is similar to incomplete Cholesky preconditioning; Gaussian elimination is performed approximately so

the resulting factors have much less fill. The novelty derives from using random sampling to produce the most useful fill in the factors. Matrix entries (edges) are split into parallel edges each owning fractional edge weight. When a variable (vertex) is eliminated, the algorithm randomly samples from this pool of fractional edges to generate partial fill in the factors. The approximate factorization computes in $O(m \log^3 n)$ time a diagonal matrix D, a lower triangular matrix L with $O(m \log^3 n)$ entries, and a permutation π such that with probability $1 - \frac{1}{poly(n)}$, the resulting factors approximate the original system with $1/2\mathcal{L} \preccurlyeq P_\pi L D L^T P_\pi^T \preccurlyeq 3/2\mathcal{L}$. Used inside a simple iterative method known as iterative refinement, they present a Laplacian solver that runs in time $O(m \log^3 n \log(1/\epsilon))$. Promising experimental results of this technique have been claimed by some researchers, but to our knowledge these results have not been published.

Chapter 3

Empirical Evaluation of Existing Algorithms

In this chapter, we present our empirical comparison of existing Laplacian solvers, work done in collaboration with Erik Boman at Sandia National Laboratories. For our experiments, we chose the Laplacian solver implementations available in the Trilinos software package developed at Sandia [47, 48]. These include a direct solver based on Cholesky decompositions as well as an iterative conjugate gradient solver with a variety of single-level and multilevel preconditioners. Descriptions of the underlying algorithms can be found in Section 2.1. While there are other Laplacian solver codes available, the goal of this work was to compare methods, not implementations. We consider the Trilinos solvers to be representative of the state of the art black box solvers available to a scientist solving $\mathcal{L}x = b$ with little information about the structure of $\mathcal{L}$. Much of this chapter is taken from our paper *An empirical comparison of graph Laplacian solvers* [59].

3.1 Test Graphs/Matrices

For these experiments we only used graphs/matrices originally stored as Laplacians, or adjacency matrices easily converted to Laplacians. We used graphs possessing between 10 thousand edges and 10 million edges. Larger graphs were left for future parallel

experiments. We used only the largest connected component of each graphs to ensure nullspace dimension of one. We include tables of test graph info at the end of this chapter.

We compiled four sets of test graphs/matrices for these experiments from three different sources. The first source was the University of Florida (UF) sparse matrix collection [60]. We divided these graphs into two categories, 2D/3D mesh-like structural problems, and graphs with irregular degree distributions that come from web or citation networks. For the most part these details are specified in the UF collection. There is a set of graphs from the DIMACS10 challenge subcollection which appear to have 2D/3D structure even though they are not classified as such that we included under 2D/3D graphs. Only unweighted adjacency matrices from the UF collection were used and converted to Laplacians. In some cases directed graphs were symmetrized by adding the transpose. Tables 3.3-3.2 include details of the UF regular and irregular degree graphs respectively.

The second set of graphs was the Block Two-Level Erdös Rényi (BTER) graphs generated with the Feastpack graph generator [61, 62]. This generator was designed to produce graphs with degree distributions and clustering properties similar to the real world networks included in the irregular UF set. Feastpack generates a graph based on the following input parameters: approximate number of vertices, average degree, maximum degree, maximum clustering coefficient, and global clustering coefficient. It gives a choice between a generalized log normal distribution and a discrete power law distribution; we always choose the former. We attempted to generate graphs with realistic parameters as described by Kolda et al. [61], modeling them after the graphs we used from the UF irregular test set. We did this to compare performance between the original graphs and the synthetic graphs designed to model them. Table 3.4 includes details of these BTER graphs.

The third set of graphs were generated from image segmentation problems. An image can be considered as a graph by treating pixels as vertices, with positive weighted edges

between them representing dissimilarity. We used Felzenszwalb and Huttenlocher's image segmentation code [63] to produce graphs from images. Graphs were named after the subject of the original image: cats, cities, food, and space images. These graphs are all 9 point meshes. Their interest for our experiments was not their structure, but rather their edge weights as this was our only test set of weighted graphs. Table 3.5 includes details of these image segmentation graphs.

3.2 Experimental Design

We performed all iterative solver experiments using the PCG implementation inside the Trilinos linear solver package Belos [64]. We used left preconditioned conjugate gradient for all the PCG experiments. We used the Ifpack2 package to create the single-level preconditioners for PCG. These include Jacobi and symmetric Gauss-Seidel relaxation preconditioners, as well as an incomplete LU factorization preconditioner ILU(0). For Laplacians, incomplete Cholesky is a better incomplete factorization, but this was not available in Trilinos. We also include results from our own Ifpack2 implementation of a maximum-weight spanning tree preconditioner. For multilevel preconditioning, we used the algebraic multigrid inside the MueLu package [65, 66]. Note that MueLu can also be used as standalone solver, but we used it as preconditioner inside PCG. We contributed an adapter to the Cholmod library [67] inside the Amesos2 [64] direct solver package, which we used to find Cholesky factorizations for performing direct solves. Our experiments were conducted with Mirasol, a large shared memory computer, at the Georgia Institute of Technology.

All solutions were found to within a residual tolerance of 10^{-9}. We generated the right hand side vector b for all experiments by first selecting a random left hand side x and multiplying by the Laplacian. We could have instead generated a random b and projected

it out of the nullspace. The number of iterations can vary significantly (hundreds of iterations) between solves (especially for the single-level preconditioners) so we used the average timing results of solving 10 right hand sides.

There are many options available in the MueLu package. We performed some initial parameter tuning on each test set using smoothed vs. unsmoothed aggregation, and default vs. degree based orderings. We selected the best result for comparisons to other solvers. On the irregular UF graphs and the BTER graphs, unsmoothed aggregation performs better than smooth aggregation. The default ordering is slightly faster for the irregular UF graphs and the degree based ordering is slightly faster for the BTER graphs. On the mesh-like UF graphs and the image segmentation graphs, smoothed aggregation performs better, with the default ordering performing slightly better for both. We experimented with different smoothers and symmetric Gauss-Seidel performed the best for all problem sets. We use the default coarse grid solver KLU2. The max coarse grid size was set to 1000 vertices.

There are various performance metrics for each solver, but here we focus on the setup time (one-time cost) and per-solve time (every-time cost). The setup time is the cost to do any preprocessing unique to a solution method, which excludes intitializing the matrix. The per-solve time is the time after the setup phase that Belos or Amesos2 takes to solve a right hand side using the solution technique. In many situations a user will only care about the sum of these two costs, but in other situations an expensive one-time setup cost might be amortized over the cost of many right hand solve times.

For the direct solver Cholmod, singularity is handled by adding 1 to the first diagonal entry. For the iterative methods, singularity is handled by modifying PCG to project the solution against the nullspace inside the solver at every iteration. The iterative methods could also handle singularity by perturbing the first diagonal entry. We performed experiments using both techniques to determine the best method and summarize the

results in Table 3.1. This table compares the ratio of the diagonal modification per-solve time over the per-solve time of the modified PCG, averaged over all the graphs for each test set. The modified PCG yields better results than a diagonal perturbation, so we chose that method.

Graph Category	Jacobi	SGS	ILU(0)	MST	MueLu
Irregular UF	1.45	1.22	1.41	1.41	1.02
2D/3D mesh-like UF	1.22	1.02	1.23	1.23	1.12
BTER	1.35	1.01	1.39	1.42	1.26
Image Segmentation	1.85	1.29	1.09	1.71	1.40

Table 3.1: Per-solve time performance comparison for handling matrix singularity. Ratio of per-solve time cost using diagonal modification over the per-solve time cost of modified PCG, averaged over all graphs in each test set.

Some of the solvers failed to converge on some test matrices. While we are interested in why these failures occur, we omit an in-depth analysis of those failure cases.

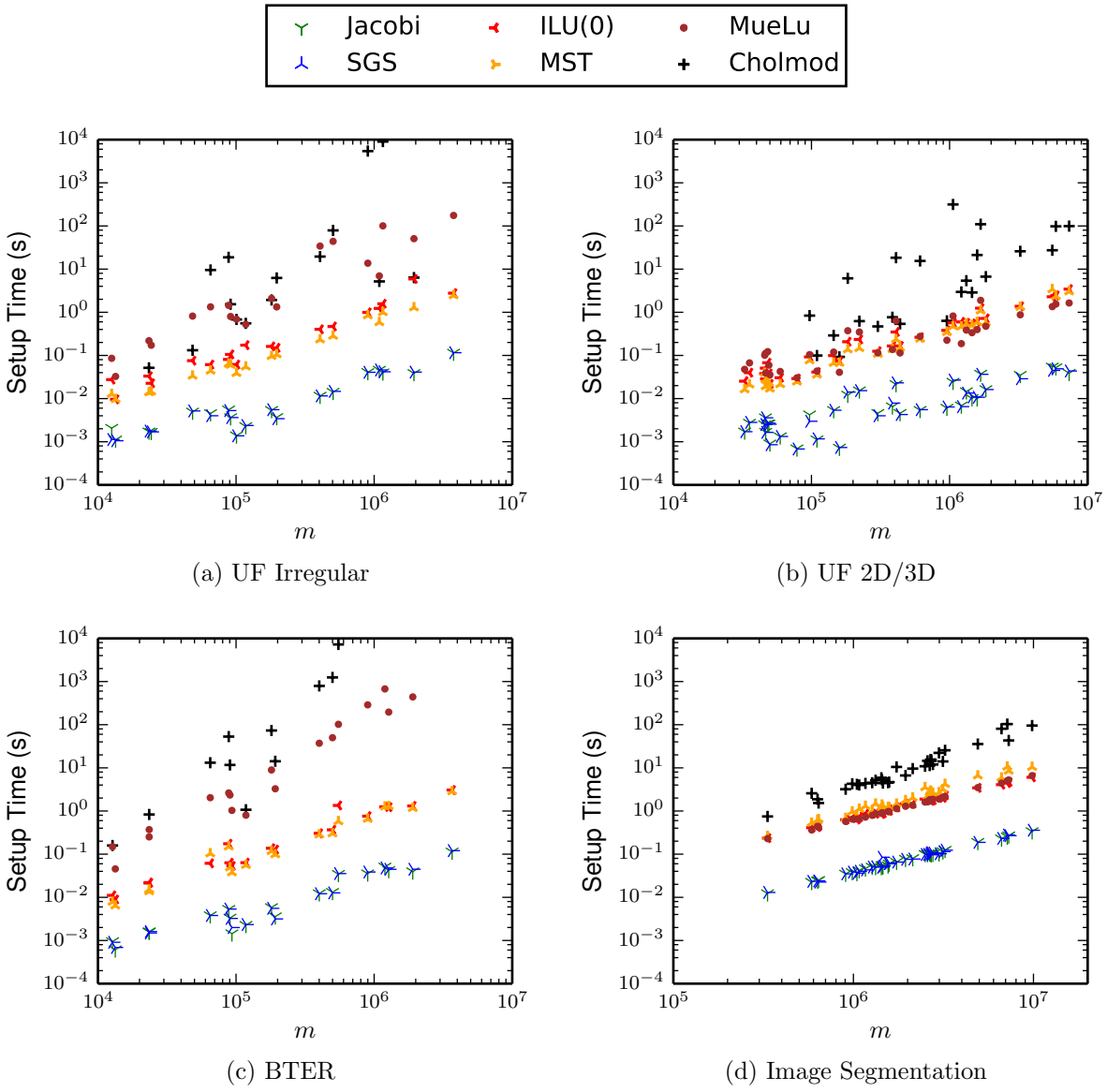
3.3 Experimental Results

3.3.1 General Performance Trends

The one-time setup cost results for every graph, in each test set, are shown as a function of graph size in edges (Figure 3.1). The peak memory usage is also shown as a function of graph size in edges (Figure 3.2).

The smallest eigenvalue of the singular systems is zero, but we modified the algorithm to stay out of the nullspace so the condition number depends on the second eigenvalue

$$\kappa(\mathcal{L}) = \frac{\lambda_n(\mathcal{L})}{\lambda_2(\mathcal{L})}.$$



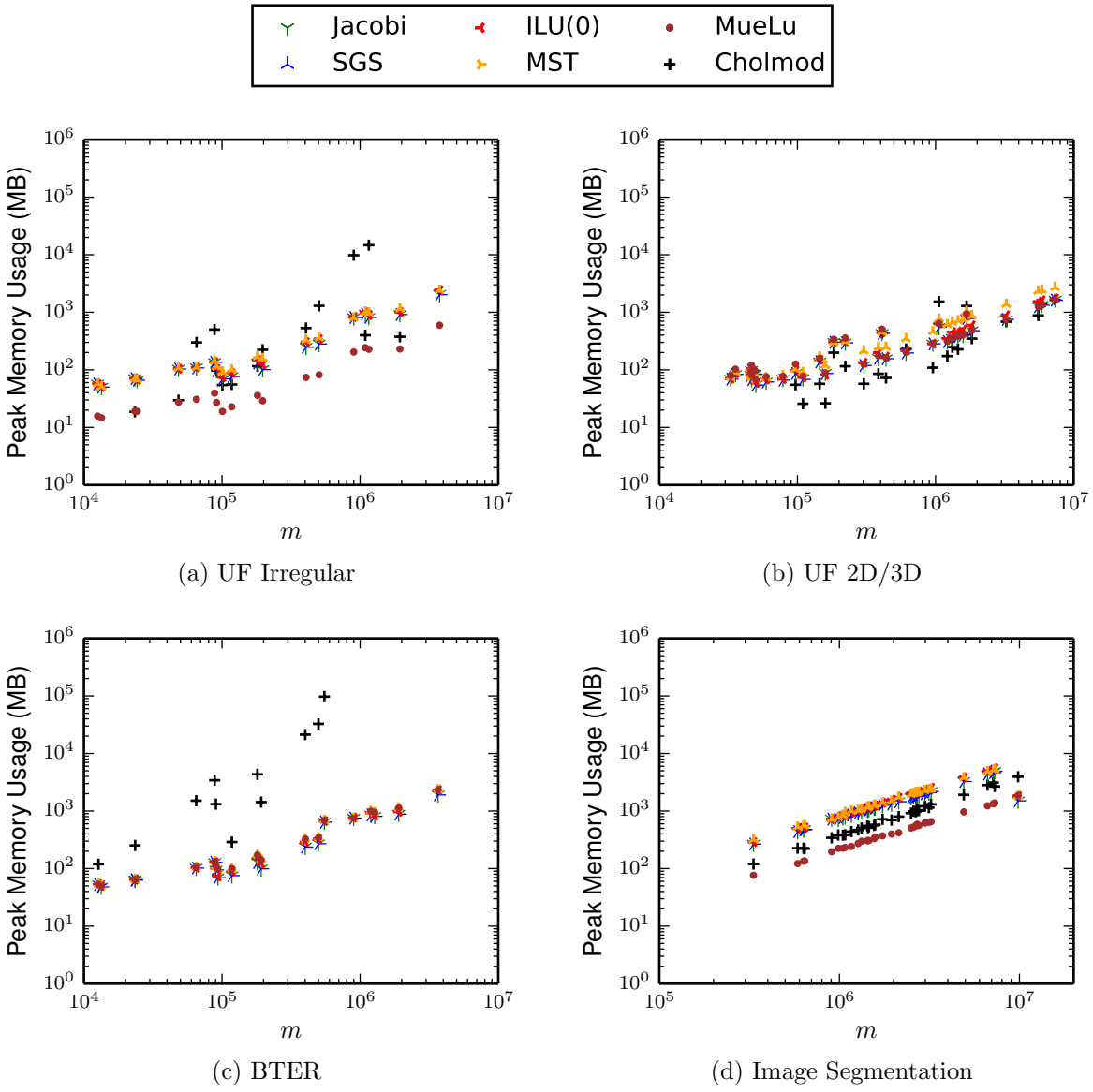
(a) UF Irregular

(b) UF 2D/3D

(c) BTER

(d) Image Segmentation

Figure 3.1: Setup time of all solvers used on each set of problems, plotted as a function of the graph size in edges of each problem.



(a) UF Irregular

(b) UF 2D/3D

(c) BTER

(d) Image Segmentation

Figure 3.2: Peak memory usage in megabytes of all solvers used on each set of set problems, plotted as a function of the number of edges of each problem.

As we expect this to be a good measure of the difficulty of a problem, we plot iteration performance results for the iterative solvers against the square root of the condition number (though to be more correct each solver performance is bounded in terms of the condition number of the preconditioned system). Figure 3.3 shows the iteration performance of every graph, separated by test set.

The per-solve time is proportional to the time to perform every iteration, times the number of iterations. The time to perform an iteration is proportional to the number of nonzeros in the matrix, or edges in the graph. Thus we plot per-solve time performance against the number of edges times the condition number. Figure 3.4 shows per-solve times of every graph, separated by test set.

3.3.2 Total Time Performance Profiles

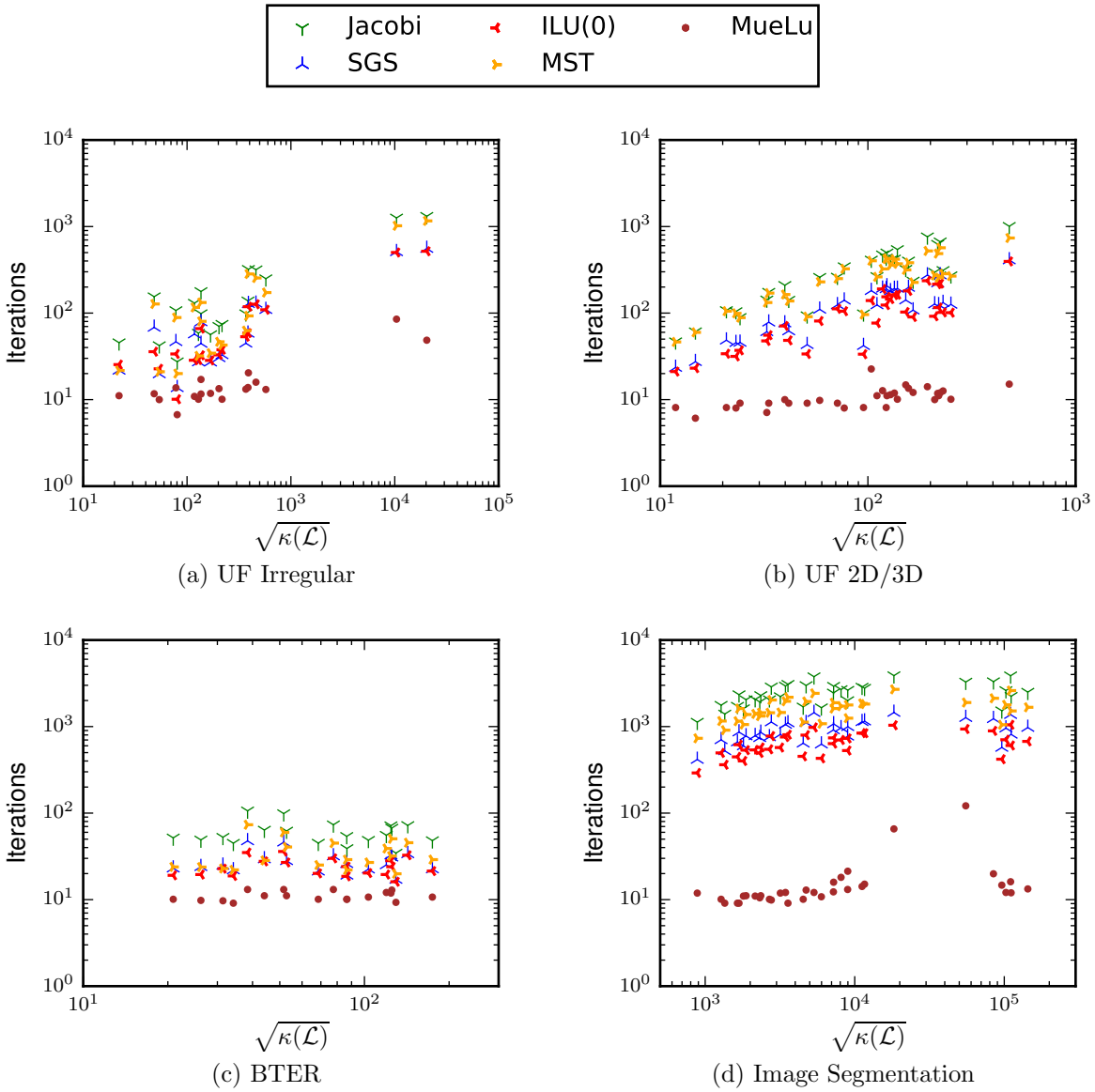
Performance profiles [68] are a useful tool for comparing the performance of multiple solvers across many test problems. For each solver s and each problem p, we define a function

$$\rho(p, s) = \frac{\text{performance of } s \text{ on } p}{\text{best performance on } p}$$

which represents the solver's performance on one problem relative to the best performance of any solver on that problem. Then for each solver we define a cumulative distribution function

$$P_s(\tau) = \frac{\#\text{ of problems in test set where } \rho(p, s) \leq \tau}{\#\text{ of problems in test set}}.$$

These distributions help compare solvers when performance varies between many problems, and some solvers fail on certain test problems entirely. Figure 3.5 shows performance profiles of the total time, the time to setup the solver and solve one right hand side, separated by test set.



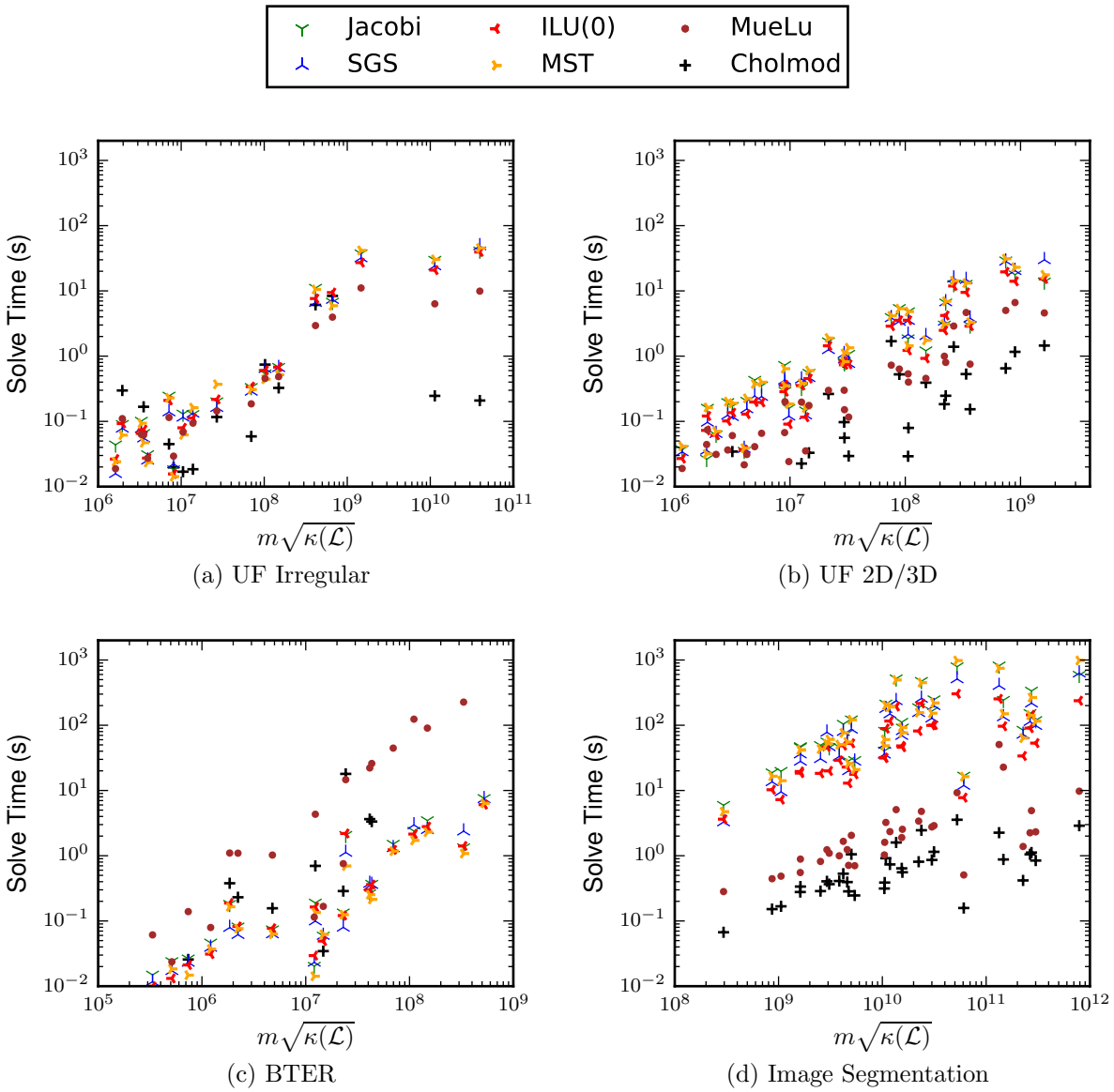
(a) UF Irregular

(b) UF 2D/3D

(c) BTER

(d) Image Segmentation

Figure 3.3: Iterations of all iterative solvers used on each set of set problems, plotted as a function of the square root of the condition number of each problem.



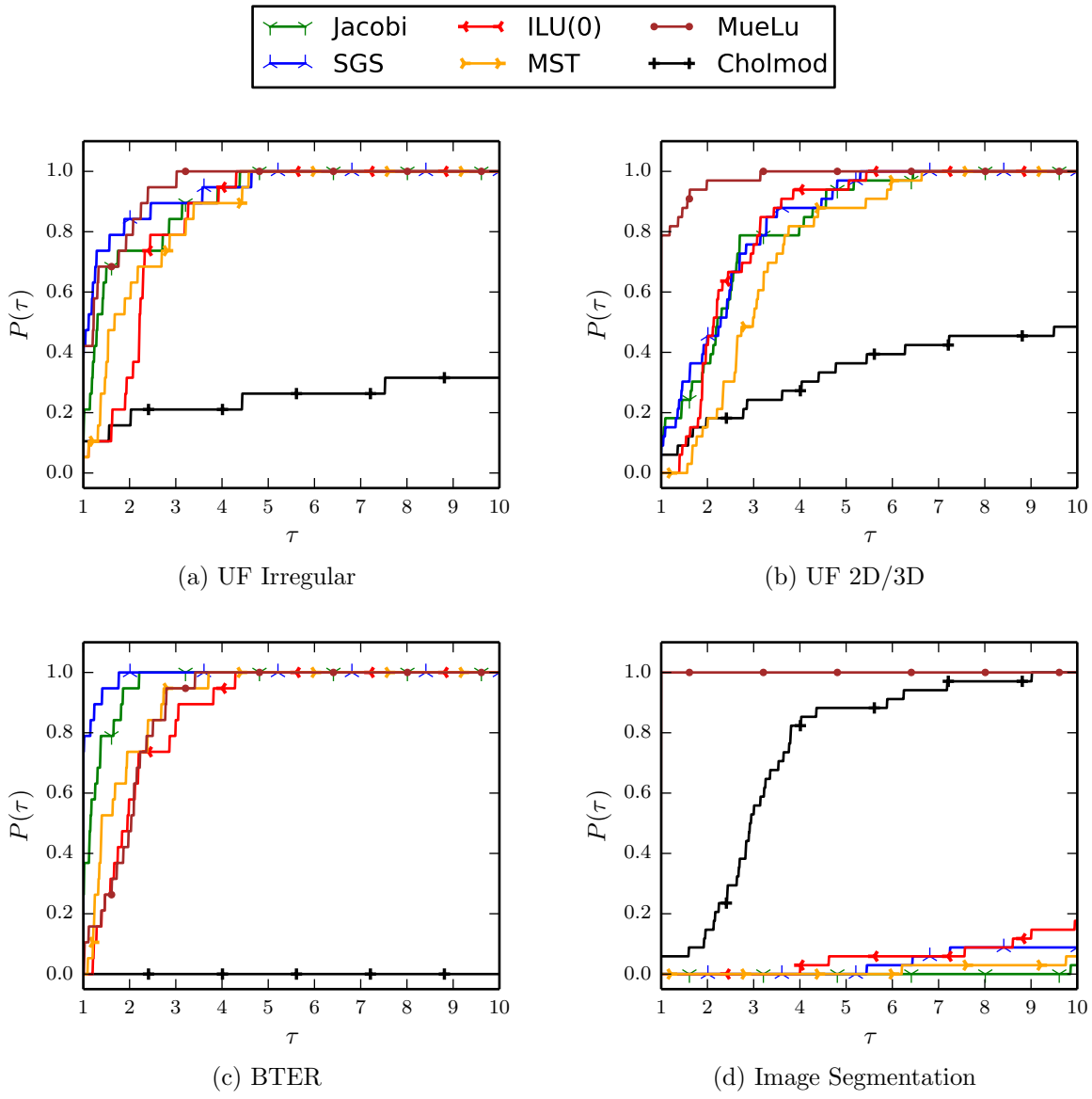
(a) UF Irregular

(b) UF 2D/3D

(c) BTER

(d) Image Segmentation

Figure 3.4: Per-solve time of all solvers used on each set of set problems, plotted as a function of the square root of the condition number times the number of edges of each problem



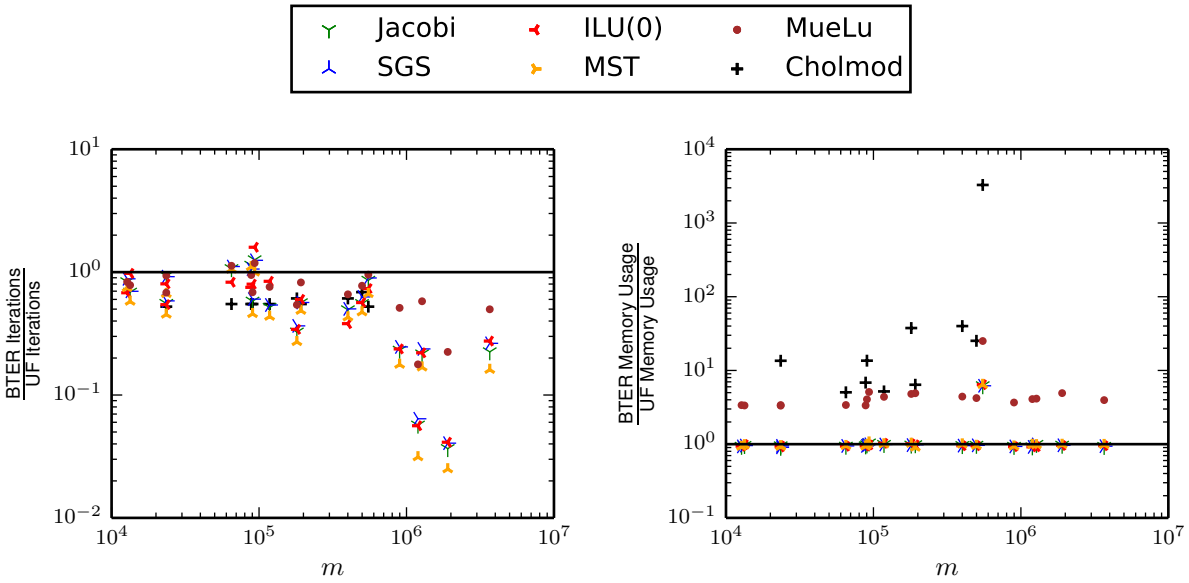
(a) UF Irregular

(b) UF 2D/3D

(c) BTER

(d) Image Segmentation

Figure 3.5: Total (setup plus per-solve) time performance profile of all solvers on each set of problems. $P(\tau)$ is the probability (over problems) that a solver is within a factor of τ of the best solver on this test data set.



(a) Iterations

(b) Peak Memory Usage

Figure 3.6: Comparison of UF Irregular graphs to their BTER replicas. The performance ratio is shown for iteration count and peak memory usage.

3.3.3 Comparing Synthetic and Collected Graphs

As we are experimenting with the synthetic BTER set, which is designed to reproduce the degree distribution and clustering behavior of some of real world networks in the irregular UF set, we directly compare the results of the web graphs with their BTER counterparts. Figure 3.6 shows the iteration and peak memory usage performance ratios for these problems.

3.4 Analysis of Experimental Results

3.4.1 General Performance Trends

The setup time and peak memory usage plots (Figures 3.1-3.2) indicate that the direct solver Cholmod will have trouble on larger graphs, as it grows at a faster rate than the iterative methods on both plots. In fact, Cholmod fails with memory errors

on some of the largest BTER graphs. However, apart from these failures, the memory usage and setup cost are not enough to rule them out completely on graphs of this size. MueLu memory usage also grows at a large rate, more so for the irregular graphs where standard aggregation could create nodes with high degree. The incomplete factorization of ILU(0) and the MST search and subsequent complete factorization make these single-level preconditioners more expensive to compute than symmetric Gauss-Seidel or Jacobi. However, increase in memory usage and setup time scale about the same for all the single-level preconditioners.

The iteration plots (Figure 3.3) indicate little change in iteration count on the BTER graphs, with a steady increase in iteration count for the single-level preconditioners on the other graphs. Using MueLu as a preconditioner results in relatively steady, low number of iterations (often on the order of 10), except on the irregular UF graphs where it does noticeably grow. The weighted image segmentation problems are the most ill-conditioned, and the single-level preconditioners take several iterations on these problems. However, single-level preconditioner iterations are also high on several problems in the irregular UF test set. The BTER graphs are the best conditioned, with the largest condition numbers being lower than several of the UF irregular graphs.

The per-solve time plots (Figure 3.4) indicate that the Cholmod direct solver typically has the best per-solve time performance, except on the BTER test set. If direct factorization succeeds without running out of memory, then the setup cost might be worth it for multiple solves. On the structured 2D/3D UF graphs and image segmentation graphs, MueLu comes close to the per-solve performance of the direct solver. It does not perform as well on the irregular problems, with very bad performance on the BTER graphs. MueLu is not optimized for graphs with irregular degree distribution and might need to incorporate ideas from the Lean Algebraic Multigrid solver [10] or Napov and Notay [36]. The single-level preconditioners are competitive with MueLu on the irregular UF graphs.

On the BTER graphs, the single-level preconditioners have the best performance.

3.4.2 Total Time Performance Profiles

We now examine the total solve time performance profiles which help indicate the best solver for a problem if a user only wants to setup and solve one right hand side. The cheap single-level preconditioners, Jacobi and symmetric Gauss-Seidel had the best performance in our experiments with the irregular UF problems (shown in Figure 3.5(a)). These were followed by the more expensive ILU(0) and spanning tree preconditioners. The large setup cost of both MueLu and Cholmod make them bad choices for solving one right hand side.

MueLu has the best performance in our experiments with the structured 2D/3D matrices from the UF test set (shown in Figure 3.5(b)). It is followed by the single-level preconditioners, all except MST are competitive with each other. The MST is expensive to compute, and the per-solve times are more expensive. Despite having a much faster per-solve time, Cholmod setup cost makes it the worst choice. Compared to the irregular UF graphs, on the mesh-like UF graphs the multilevel MueLu preconditioner outperforms the single-level preconditioners. From the performance profile it is difficult to conclude the best single-level preconditioner.

On the BTER test set (shown in Figure 3.5(c)), relative solver performance is similar to the irregular UF graph results. The single-level preconditioners, Jacobi and SGS are cheap and effective, giving them the best performance on this test set. These are followed by the more expensive ILU(0) and MST. MueLu and Cholmod have even more difficulty on these problems than on the irregular UF problems. Since Cholmod runs out of memory on the larger of these problems, the difficulty might be in large factor fill, both for Cholmod and the MueLu coarse grid solver.

MueLu has the best performance by far in our experiments with the image segmentation problems (shown in Figure 3.5(d)) The per-solve times for the single-level methods are too large, and the Cholmod setup cost is too large. However, Cholmod did outperform the single-level preconditioners on these test problems, despite the setup cost. This speaks to something stronger than these single-level preconditioners being needed on matrices with high enough condition number.

3.4.3 Comparing Synthetic and Collected Graphs

We noticed a few differences between performance on irregular UF web graphs and BTER graphs modeled after them. In Figure 3.6(a) we see that the number of iterations required is typically less for the BTER graphs, sometimes by a few orders of magnitude. In Figure 3.6(b) we see that the memory usage of the direct solver, and the coarse grid direct solvers is much higher for the BTER graphs. In fact this plot is missing data points where the factorization failed altogether on the BTER problems. BTER graphs were designed to replicate degree distribution and community structure, not spectral properties such as eigenvalues and condition numbers. It is clear from the above plots that the BTER replicas are better conditioned than the web network graphs they are supposed to model. This must manifest itself in some structural difference between the originals and replicas to cause different performance on these problems.

3.5 Discussion

We performed an initial experimental evaluation of existing Laplacian solvers on a variety of different problems. While there are many directions to expand this work, there are a few points we can already conclude. Relative solver performance is fairly consistent within each test set, but very different between test sets. On the problems with mesh-like

structure, the 2D/3D UF and image segmentation sets, the multilevel MueLu solver has the best performance. Single-level preconditioners are currently the best choice on the irregular UF and BTER problem sets. The direct solver Cholmod can solve a right hand side very quickly once the expensive factorization setup cost is done, perhaps making it useful for multiple right hand solves but not a single solve.

Solver iterations and per-solve time scale well as a function of condition number. The weighted image segmentation problems have the largest condition number and are thus more difficult problems to solve. In contrast, the BTER graphs have low condition number and are easier to solve. Perhaps because these problems are easier, the cheapest single-level preconditioners are all that is needed. This makes it difficult to answer the question of where techniques from more theoretical solvers could be useful in practice, as multigrid performs very well on mesh-like graphs, and the irregular graphs are even less difficult problems. We wonder if we could generate a set of irregular graphs that are more ill-conditioned, and whether or not these cheap single-level preconditioners will still be optimal.

We notice different performance behavior between the synthetic BTER graphs and the irregular UF graphs they are supposed to model, despite having similar degree distributions and clustering coefficients. In some sense the BTER problems are easier for the iterative solvers as they are better conditioned. On the other hand the BTER graphs seem to give Cholmod and MueLu a more difficult time, due to increased fill during factorization. There must be some structural differences between these graphs that impact condition number and factorization fill. The nature of these structural differences and whether or not they are a weakness of the BTER model are open questions. How graph structure affects condition number is a question we will examine more in Chapter 5.

Graph	Collection	Nodes	Edges
as-735	SNAP	6.47K	12.6K
ca-GrQc	SNAP	4.16K	13.4K
Oregon-1	SNAP	11.2K	23.4K
PGPgiantcompo	Arenas	10.7K	24.3K
as-22july06	Newman	23.0K	48.4K
p2p-Gnutella24	SNAP	26.5K	65.4K
p2p-Gnutella30	SNAP	36.6K	88.3K
ca-CondMat	SNAP	21.4K	91.3K
wiki-Vote	SNAP	7.07K	101K
ca-HepPh	SNAP	11.2K	118K
emailEnron	SNAP	33.7K	181K
ca-AstroPh	SNAP	17.9K	197K
soc-Epinions1	SNAP	75.9K	406K
soc-Slashdot0902	SNAP	82.2K	504K
citationCiteseer	DIMACS10	268K	1.16M
amazon0302	SNAP	252K	890K
web-NotreDame	SNAP	326K	1.09M
web-Stanford	SNAP	255K	1.94M
amazon-2008	LAW	735K	3.52M

Table 3.2: Irregular UF graphs/matrices used, along with collection they came from.

Graph	Collection	Nodes	Edges
fe_4elt2	DIMACS10	11.1K	32.8K
L9	AG-Monien	18.0K	35.6K
barth5	Pothen	15.6K	45.9K
stufe10	AG-Monien	24.0K	46.4K
shuttle_eddy	Pothen	10.4K	46.6K
cti	DIMACS10	16.8K	48.2K
fe_sphere	DIMACS10	16.4K	49.2K
nasa4704	Boeing	4.70K	50.0K
skirt	Pothen	7.93K	59.4K
lock3491	HB	3.42K	78.5K
copter1	GHS_psdef	17.2K	96.9K
man_5976	HB	5.88K	110K
pwt	Pothen	36.5K	145K
cegb2919	HB	2.86K	159K
tandem_dual	Pothen	94.1K	183K
ford2	GHS_psdef	100K	222K
bcsstk29	HB	13.8K	302K
struct3	Rothberg	32.1K	386K
fe_ocean	DIMACS10	143K	410K
tube1	TKK	21.5K	438K
pli	Li	18.4K	612K
trdheim	DNVS	22.1K	957K
wave	AG-Monien	156K	1.06M
tsyl201	DNVS	20.7K	1.22M
pct20stif	Boeing	52.3K	1.32M
srb1	GHS_psdef	54.9K	1.45M
3dtube	Rothberg	45.3K	1.58M
m14b	DIMACS10	215K	1.68M
s4dkt3m2	TKK	90.4K	1.83M
gearbox	Rothberg	108K	3.25M
fcondp2	DNVS	202K	5.55M
troll	DNVS	213K	5.89M
pkustk14	Chen	152M	7.34M

Table 3.3: 2D/3D mesh-like UF graphs/matrices used, along with collection they came from.

Graph	Nodes	Edges
BTER_amazon0302	238K	2.03M
BTER_amazon-2008	677K	8.02M
BTER_as-22july06	229K	1.33M
BTER_as-735	5.60K	31.1K
BTER_ca-AstroPh	17.0K	402K
BTER_ca-CondMat	19.7K	200K
BTER_ca-CrQc	3.59K	30.3K
BTER_ca-HepPh	10.3K	246K
BTER_citationCiteseer	10.3M	2.56M
BTER_email-Enron	31.3K	392K
BTER_Oregon-1	9.71K	56.7K
BTER_p2p-Gnutella24	23.4K	153K
BTER_p2p-Gnutella30	33.4K	210K
BTER_PGPgiantcompo	9.22K	56.2K
BTER_soc-Epinions1	70.2K	871K
BTER_soc-Slashdot0902	76.8K	1.08M
BTER_web-NotreDame	76.8K	2.47M
BTER_web-Stanford	295K	4.10M
BTER_wiki-Vote	6.75K	193K

Table 3.4: BTER graphs/matrices used.

Graph	Nodes	Edges
cat1	693K	2.76M
cat2	247K	986K
cat3	1.23M	4.91M
cat4	333K	1.33M
cat5	161K	641K
cat6	1.80M	7.16M
city1	687K	2.65M
city2	362K	1.44M
city3	786K	3.14M
city4	674K	2.70M
city5	680K	2.70M
city6	488K	1.95M
city7	391K	1.56M
city8	827K	1.17M
city9	2.46M	9.83M
city10	270K	1.08M
city11	647K	2.58M
city12	396K	1.58M
city13	627K	2.50M
city14	147K	586K
city15	83.8K	333K
food1	677K	2.70M
food2	535K	2.14M
food3	160K	632K
food4	450K	1.74M
food5	227K	906K
food6	813K	3.23M
food7	1.95M	7.32M
space1	319K	1.27M
space2	748K	2.99M
space3	366K	1.46M
space4	1.66M	6.65M
space5	358K	1.43M
space6	263K	1.05M

Table 3.5: Image segmentation graphs/matrices used.

Chapter 4

KOSZ, Cycle Toggling, and Heavy Path Graphs

We include our experimental work on the KOSZ algorithm in this chapter. This algorithm was first described in a paper by Kelner, Orecchia, Sidford, and Zhu [6], which is why it is often referred to as the KOSZ algorithm, or simply the Kelner method. In our work, we have also referred to this method as dual randomized Kaczmarz [55, 56, 57], because it is useful to think of the algorithm performing Kaczmarz projections in the dual space, and because we desired a slightly more general term for algorithms based on KOSZ with differing implementation details. KOSZ is the more common terminology so we use that here. We also examine a core primitive of this algorithm, the update of flow information on cycle edges, which we refer to as *cycle toggling*. To study cycle toggling, we designed an interesting class of model graphs called *heavy path graphs*, that we believe to be an interesting test case.

This work was done in collaboration with Erik Boman, Garry Miller, Richard Peng, Haoran Xu, and Shen Chen Xu. Much of this work can be found in our papers *Evaluating the dual randomized Kaczmarz Laplacian linear solver* [69] and *An empirical study of cycle toggling based Laplacian solvers* [70].

4.1 Algorithm Background

The inspiration for the algorithm proposed by Kelner et al. is to treat graphs as electrical networks with resistors on the edges. For each edge, the weight is the inverse of the resistance ($r_e = 1/w_e$). We can think of vertices as having an electrical potential and a net current at every vertex, and define vectors of these potentials and currents as v and f respectively. These vectors are related by the linear system $\mathcal{L}v = f$. Solving this system is equivalent to finding the set of voltages that satisfies the net "injected" currents. Kelner et al.'s algorithm solves this problem with an optimization algorithm in the dual space, which finds the optimal currents on all of the edges subject to the constraint of zero net voltage around all cycles. Note that in circuit theory this is Kirchoff's Voltage Law. Their method uses Kaczmarz projections [55] to adjust currents on one cycle at a time, iterating until convergence.

We will also refer to the Primal Randomized Kaczmarz (PRK) method that applies Kaczmarz projections in the primal space [56]. One sweep of PRK performs a Kaczmarz projection with every row of the matrix. Rows are taken in random order at every sweep.

KOSZ iterates over a set of *fundamental cycles*, cycles formed by adding individual edges to a spanning tree T. The fundamental cycles are a basis for the space of all cycles in the graph [71]. For each off-tree edge e, we define the resistance R_e of the cycle C_e that is formed by adding edge e to the spanning tree as the sum of the resistances around the cycle,

$$R_e = \sum_{e' \in C_e} r_{e'}$$

which is thought of as approximating the resistance of the off-tree edge r_e. KOSZ chooses fundamental cycles randomly, with probability proportional to R_e/r_e.

The performance of the algorithm depends on the sum of these approximation ratios,

a property of the spanning tree called the *tree condition number*

$$\tau(T) = \sum_{e \in E \setminus T} \frac{R_e}{r_e}.$$

The number of cycle updates KOSZ requires is proportional to the tree condition number. KOSZ uses a particular type of spanning tree with low tree condition number, called a *low stretch tree*. Specifically, the original KOSZ description used the one described by Abraham and Neiman [42] with $\tau = O(m \log n \log \log n)$. The work of one cycle update is naively the cycle length, but can be reduced to $O(\log n)$ with a fast data structure, yielding $O(m \log n^2 \log \log n)$ total work.

4.1.1 Related Work

As the KOSZ algorithm is a recent and theoretical result, there are few existing implementations or performance results. Hoske et al. implemented the KOSZ algorithm in C++ and did timing comparisons against unpreconditioned CG on two sets of generated graphs [72]. They concluded that the solve time of KOSZ does scale nearly linearly. However, several factors make the running time too large in practice, including large tree stretch and cycle updates with unfavorable memory access patterns. They cited experimental results by Papp [73], which suggest that the theoretically low stretch tree algorithms are not significantly better than maximum-weight spanning trees in practice, at least on relatively small graphs.

4.2 Empirical Comparison of KOSZ, PCG, and PRK

4.2.1 Experimental Design

Our initial study of KOSZ measured performance in terms of work instead of time, and used a somewhat more diverse graph test set than Hoske et al. [72]. We implemented the algorithm in Python with Cython to see how it compared against PCG (preconditioned with Jacobi diagonal scaling) and PRK. However, we did not implement a low stretch spanning tree. Instead we used a low stretch heuristic that ranks and greedily selects edges by the sum of their incident vertex degrees (a cheap notion of centrality). In practice this works well on unweighted graphs. We also did not implement the fast data structure described by Kelner et al. to update cycles in $O(\log n)$ work.

Our initial results did not include wall clock time, since our KOSZ implementation was not highly optimized. Instead we were interested in measuring the total work. For PCG work is the number of nonzeros in the matrix for every iteration, plus the work of applying the preconditioner at every iteration (number of vertices for Jacobi). For PRK the work is the number of nonzero entries of the matrix for every sweep, where a sweep is a Kaczmarz projection against all the rows of $\mathcal{L}$. As the KOSZ work will depend on data structures and implementation, we consider four different costs for estimating the work of updating a single cycle, which we refer to as cost metrics.

Metric 1. cycle length (naive)

Metric 2. $\log n$ (using fast update data structure)

Metric 3. log(cycle length) (optimistic)

Metric 4. 1 (lower bound)

The first metric charges the number of edges in the cycle, which is included because it

is the naive implementation we used for this study. The second metric, based upon the data structure described by Kelner et al., charges $\log n$ work per cycle update. This may be an overestimate when the cycle length is actually less than $\log n$. The third metric charges log(cycle length), a hypothetical update method which we do not know to exist, but is included as a hopeful estimate of a potentially better update data structure. The last metric charges unit work per cycle, which is included because we surely cannot do better than this.

Graph (Collection)	Nodes	Edges	2-core Nodes	2-Core Edges	Greedy Cycles	Probability of Selecting Greedy	Largest Cycle Length
jagmesh3 (HB)	1.09k	3.14k	1.09k	3.14k	1.92k	0.2419	77
lshp1270 (HB)	1.27k	3.70k	1.27k	3.70k	2.17k	0.4712	95
rail_1357 (Oberwolfach)	1.36k	3.81k	1.36k	3.81k	1.85k	0.2507	55
50 x 50 grid	2.50k	4.90k	2.50k	4.90k	2.40k	0.5000	120
data (DIMACS10)	2.85k	15.1k	2.85k	15.1k	7.43k	0.1760	92
100 x 100 grid	10.0k	19.8k	10.0k	19.8k	9.80k	0.5000	230
20 x 20 x 20 grid	8.00k	22.8k	8.00k	22.8k	3.57k	0.1941	122
L-9 (A-G Monien)	18.0k	35.6k	18.0k	35.6k	17.6k	0.4992	411
tuma1 (GHS_indef)	23.0k	37.2k	22.2k	36.5k	10.7k	0.0610	420
barth5 (Pothen)	15.6k	45.9k	15.6k	45.9k	29.9k	0.1765	375
cti (DIMACS10)	16.8k	48.2k	16.8k	48.2k	7.27k	0.0501	172
aft01 (Okunbor)	8.21k	58.7k	8.21k	58.7k	26.6k	0.6680	105
30 x 30 x 30 grid	27.0k	78.3k	27.0k	78.3k	8.35k	0.1399	202
wing (DIMACS10)	62.0k	122k	62.0k	122k	27.9k	0.0301	605
olesnik0 (GHS_indef)	88.3k	342k	88.3k	342k	220k	0.1327	363
tube1 (TKK)	21.5k	438k	21.5k	438k	0	0.0000	102
fe_tooth (DIMACS10)	78.1k	453k	78.1k	453k	217k	0.3673	286
dawson5 (GHS_indef)	51.5k	480k	20.2k	211k	19.8k	0.0941	165

Table 4.1: Statistics of all Mesh-like Graphs Used in Experiments

We ran experiments on all the mesh-like graphs and irregular graphs shown in Tables 4.1-4.2. Mesh-like graphs come from more traditional applications such as model reduction and structure simulation, and contain a more regular degree distribution. Irregular graphs come from electrical, road, and social networks, and contain a more irregular, sometimes exponential, degree distribution. Most of these graphs are in the

University of Florida (UF) sparse matrix collection [60]. We added a few 2D and 3D grids along with a few graphs generated with the BTER generator [61]. We removed weights and in a few cases symmetrized the matrices by adding the transpose. We pruned the graphs to the largest connected component of their 2-core, by successively removing all degree 1 vertices, since KOSZ operates on the cycle space of the graph. The difference between the original graph and the 2-core is trees that are pendant on the original graph. These can be solved in linear time so we disregarded them to see how solvers compare on just the structurally interesting part of the graph.

We solved to a relative residual tolerance of 10^{-3}. Accuracy in the solution was scarified in order to run more experiments and on larger graphs. The Laplacian matrices are singular with a nullspace dimension of one (because the pruned graph is connected). For KOSZ and PRK this is not a problem, but for PCG we must handle the non-uniqueness of the solution. Our choice for handling singularity was to remove the last row and column of the matrix. One could also choose to orthogonalize the solution against the nullspace inside the algorithm; in our experience performance results are similar.

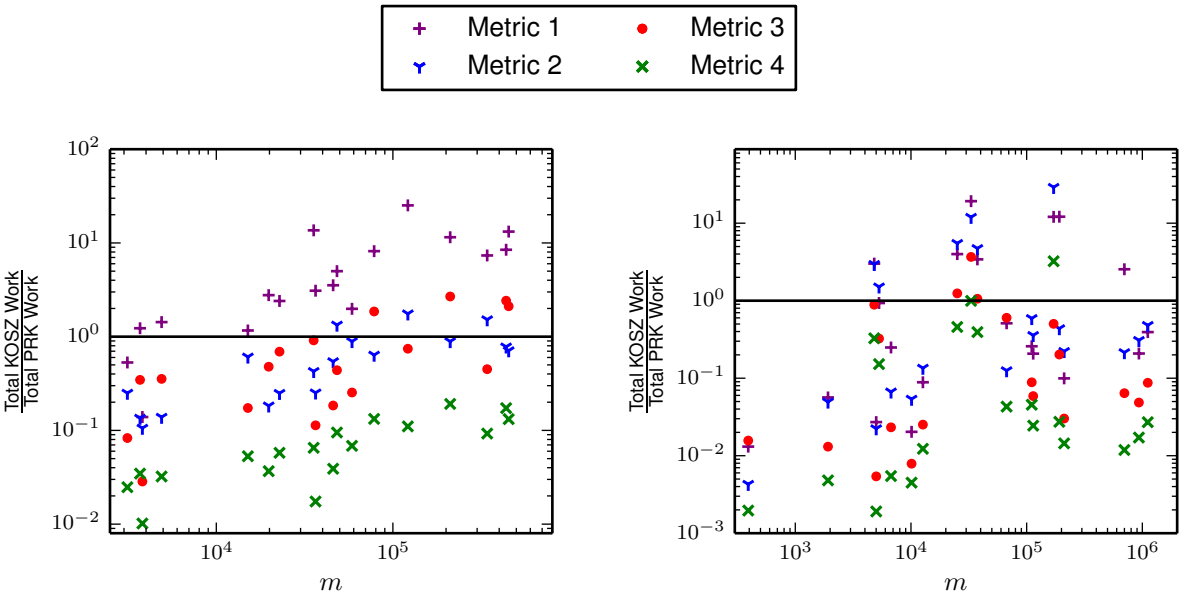
We also ran a set of PCG vs. KOSZ experiments where the convergence criteria was the error within 10^{-3}. The error can only be calculated by knowing the solution in advance, so it is hard to do this in practice. One of the interesting behaviors of the KOSZ algorithm is that, unlike PCG and PRK, convergence does not depend on the condition number of the matrix, but instead just on the tree condition number. Since higher condition number can make small residuals less trustworthy, we wondered whether convergence in the error yields different results.

Graph (Collection)	Nodes	Edges	2-core Nodes	2-core Edges	Greedy Cycles	Probability of Selecting Greedy	Largest Cycle Length
EVA (Pajek)	8.50k	6.71k	314	492	84	0.2346	18
bcspwr09 (HB)	1.72k	2.40k	1.25k	1.92k	651	0.3276	54
BTER1 $d_{avg} = 10$, $d_{max} = 30$ $cc_{max} = .3$, $cc_{global} = .1$	981	4.85k	940	4.82k	510	0.0465	18
USpowerGrid (Pajek)	4.94k	6.59k	3.35k	5.01k	1.68k	0.2997	80
email (Arenas)	1.13k	5.45k	978	5.30k	362	0.0433	11
uk (DIMACS10)	4.82k	6.84k	4.71k	6.72k	1.97k	0.2488	211
as-735 (SNAP)	7.72k	13.9k	4.02k	10.1k	3.83k	0.0822	9
ca-GrQc (SNAP)	4.16	13.4k	3.41k	12.7k	4.43k	0.2315	22
BTER2 $d_{avg} = 10$, $d_{max} = 70$ $cc_{max} = .3$, $cc_{global} = .1$	4.86k	25.1k	4.54k	24.8k	2.69k	0.0468	17
gemat11 (HB)	4.93k	33.1k	4.93k	33.1k	9.72k	0.0011	42
BTER3 $d_{avg} = 15$, $d_{max} = 70$ $cc_{max} = .6$, $cc_{global} = .15$	4.94k	37.5k	4.66k	37.2k	4.79k	0.0518	18
dictionary28 (Pajek)	52.7k	89.0k	20.9k	67.1k	20.2k	0.1410	36
astro-ph (SNAP)	16.7k	121k	11.6k	111k	13.2k	0.0786	18
cond-mat-2003 (Newman)	31.2k	125k	25.2k	114k	32.5k	0.1533	23
BTER4 $d_{avg} = 15$, $d_{max} = 30$ $cc_{max} = .6$, $cc_{global} = .15$	999	171k	999	171k	33	0.0002	7
HTC_336_4438 (IPSO)	226k	339k	64.1k	192k	32.9k	0.0339	990
OPF_10000 (IPSO)	43.9k	212k	42.9k	211k	122k	0.3146	53
ga2010 (DIMACS10)	291k	709k	282k	699k	315k	0.1466	941
coAuthorsDBLP (DIMACS10)	299k	978k	255k	934k	297k	0.1524	36
citationCiteseer (DIMACS10)	268k	1.16M	226k	1.11M	150k	0.0484	56

Table 4.2: Statistics of all Irregular Graphs Used in Experiments

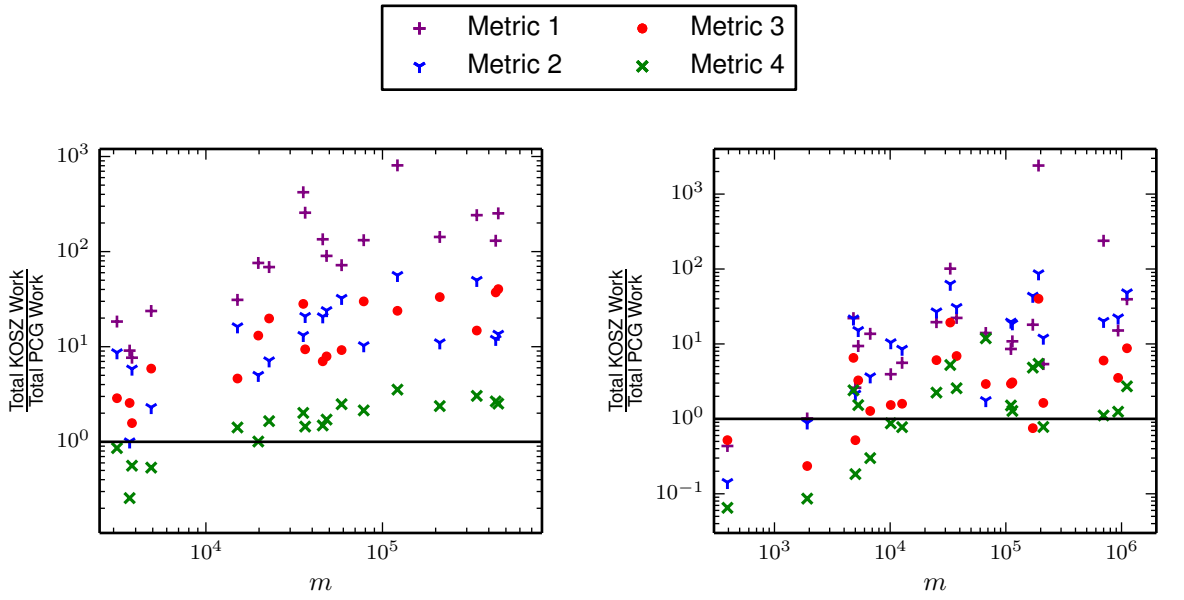
4.2.2 Experimental Results

We compared KOSZ to the other solvers by examining the ratio of KOSZ work to the work of the other solvers. We plot the ratio of KOSZ work to PRK work in Figure 4.1, separated by graph type. Each vertical set of four points are results for a single graph, and are sorted on the x axis by graph size. The four points represent the ratio of KOSZ work to PRK work under all four cost metrics. Points above the line indicate KOSZ



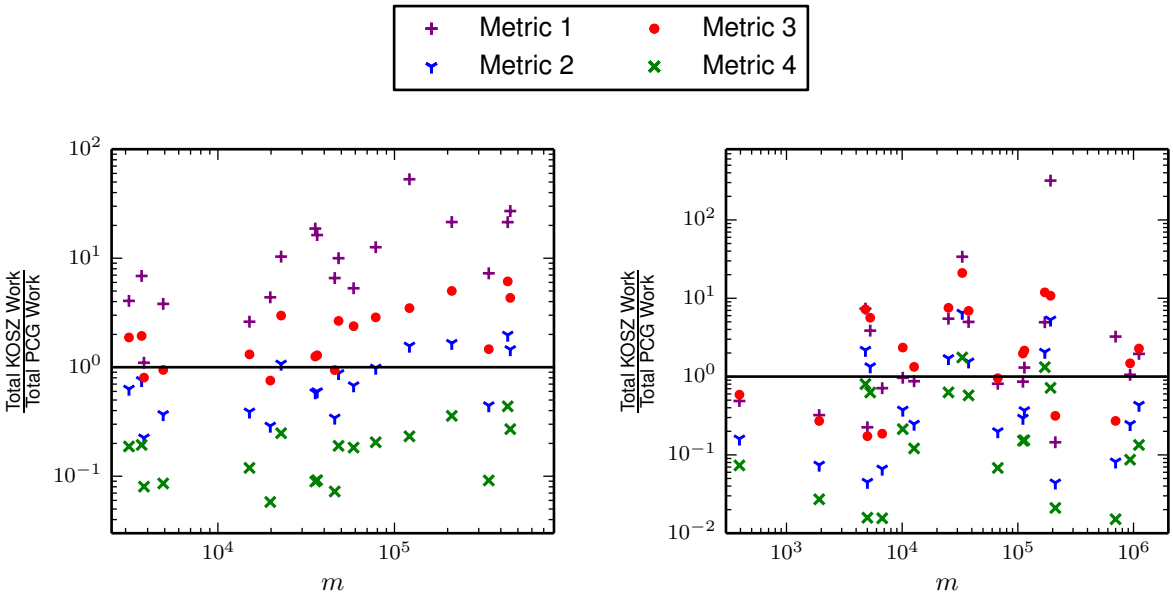
(a) Mesh-like Graphs (b) Irregular Graphs

Figure 4.1: KOSZ vs. PRK: Relative work of KOSZ to PRK work under the four cost metrics is shown (PRK is better than KOSZ at points above the line).



(a) Mesh-like Graphs (b) Irregular Graphs

Figure 4.2: KOSZ vs. PCG: Relative work of KOSZ to PCG work under the four cost metrics is shown (PCG is better than KOSZ at points above the line).



(a) Mesh-like Graphs (b) Irregular Graphs

Figure 4.3: KOSZ vs. PCG Converged to Error: Relative work of KOSZ to PCG work under the four cost metrics is shown, convergence tolerance is norm of error within 10^{-3}.

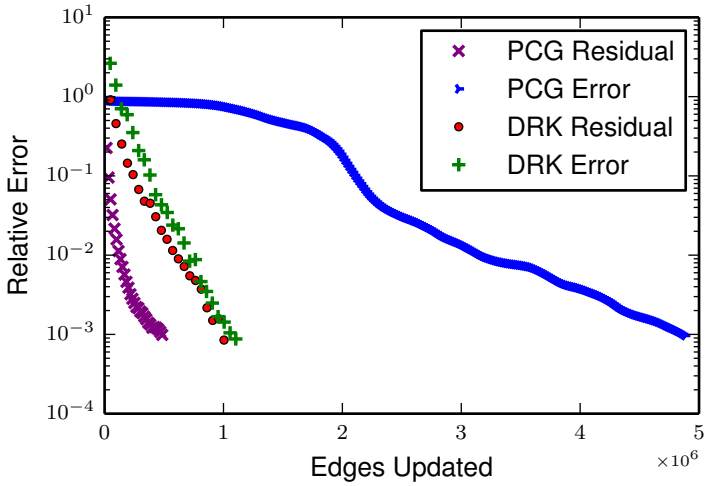


Figure 4.4: KOSZ and PCG Convergence Behavior on USpowerGrid: Residual and error are shown for both solvers over the iterations required for convergence.

performed more work while points below the line indicate KOSZ performed less work. We plot similar results for the PCG comparison in Figure 4.2. We show another set of PCG comparisons, converged to the error, in Figure 4.3.

Figure 4.4 shows an example of the convergence behavior on the USpowerGrid graph. This plot illustrates how both the error and residual behave during the solve for both PCG and KOSZ. A steeper slope indicates faster convergence. Note this only shows metric 1 work for KOSZ.

4.2.3 Experimental Analysis

In the comparison to PRK (shown in Figure 4.1), KOSZ was often better with cost metrics 3 and 4. On a few graphs, mostly in the irregular category, KOSZ outperformed PRK in all cost metrics (all the points are below the line). In the comparison to PCG (shown in Figure 4.2), KOSZ fares slightly better for the irregular graphs, but on both graph sets these results are somewhat less than promising. PCG often performed better (most of the points are above the line). Even if we unrealistically assume unit cost for cycle updates, PCG outperformed KOSZ. The performance ratios also seem to get worse as graphs get larger.

The results concerning the error (shown in Figure 4.3) are very interesting as they are quite different than those with the residual tolerance. For all of the mesh graphs, considering the error convergence makes KOSZ look more promising. The relative performance of cost metrics 3 and 4 are now typically better for KOSZ than PCG. However, PCG is still consistently better with cost metrics 1 and 2. For some of the irregular graphs, the convergence behavior is similar, but for others things look much better when considering error convergence. Informally the number of edges updated by KOSZ did not change much when switching convergence criteria, but PCG work often increased. The USpowerGrid example (shown in Figure 4.4) gives a sense of this. The residual and error decrease similarly for KOSZ, but the error curve for PCG decreases much more slowly than the residual.

4.3 Non-Fundamental Cycle Sets and Potential Parallelism

We considered ways in which KOSZ could be improved by altering the choice of cycles and their updates. Our goals are both to reduce total work and to identity potential parallelism in KOSZ. To this end we are interested in measuring the number of *parallel steps*, the longest number of steps a single thread would have to perform before convergence, maximized over all threads. Parallel steps are measured in terms of the four cost metrics described in Section 4.2.1. We also define the *span* [74], or critical path length, which is the number of parallel steps with unbounded threads.

4.3.1 Expanding the Set of Cycles

Sampling fundamental cycles with respect to a tree may require updating several long cycles which will not be edge-disjoint. It would be preferable to update edge-disjoint cycles, as these updates could be done in parallel. The cycle set we use does not need to be a basis, but it does need to span the cycle space. In addition to using a cycle basis from a spanning tree, we will use several small, edge-disjoint cycles. We expect that having threads update these small cycles is preferable to having them stand idle.

2D Grid Example

A simple example of a different cycle basis is the 2D grid graph, shown in Figure 4.5. In the original KOSZ, cycles are selected by adding off-tree edges to the spanning tree as in Figure 4.5(a). As the 2D grid graph is planar, the faces of the grid are the regions bounded by edges, and we refer to the cycles that enclose these regions as *facial cycles*. We consider using these cycles to perform updates of KOSZ, as the facial cycles span the

cycle space of a planar graph [71]. Half of these cycles can be updated at one iteration and then the other half can be updated during the next iteration, in a checkerboard fashion, as in Figures 4.5(b)(c). Furthermore, to speed up convergence, smaller cycles can be added together to form larger cycles (in a multilevel fashion) as in Figure 4.5(d).

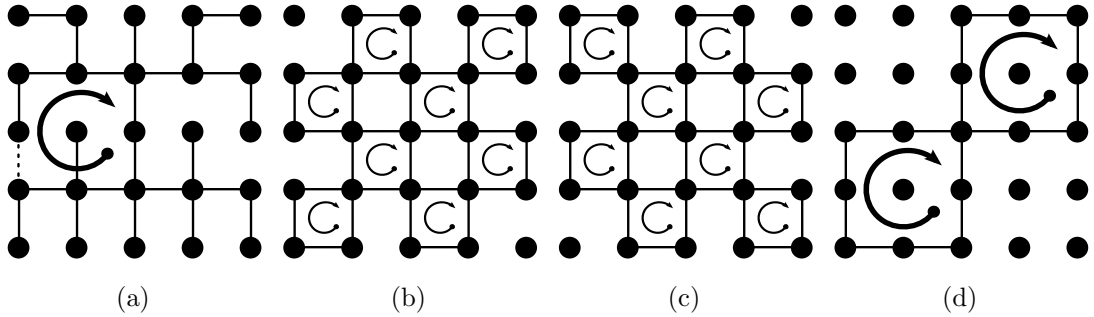


Figure 4.5: Grid Cycles: (a) Fundamental cycles are formed by adding edges to the spanning tree. (b-c) First level facial cycles are shown, grouped into edge-disjoint sets. (d) Second level facial cycles are formed by adding smaller facial cycles.

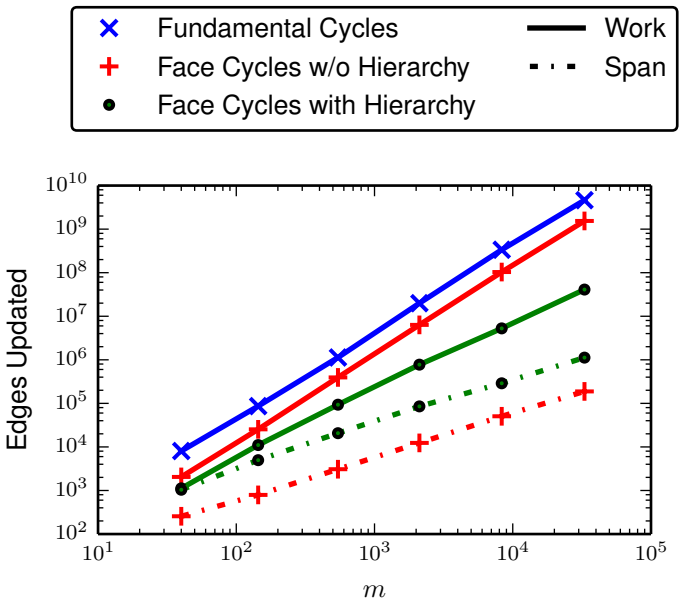


Figure 4.6: Grid Cycle Performance: Work and span of KOSZ using facial cycles and fundamental cycles for two dimensional grids of various sizes.

We implemented such a cycle update scheme using the grid facial cycles, and performed experiments to see how the facial cycles affected the total work measured in both the number of cycles updated (metric 4) and edges updated (metric 1). With the facial cycles, the span per iteration is the cost of updating two cycles at each level. We ran experiments with and without the hierarchical combination of the facial cycles against the original set of fundamental cycles. In the case of the fundamental cycles we used H trees [44], which have optimal stretch $O(\log n)$. Solutions were calculated to a residual tolerance of 10^{-6}. The accuracy here is slightly better than the rest of the experiments since these experiments were faster.

The results shown in Figure 4.6 indicate that the facial cycles improve both the work and span. Using a hierarchical update scheme reduces the total number of edges updated. However as this requires updating larger cycles it has a worse span than simply using the lowest level of cycles.

4.3.2 Extension to General Graphs

We refer to the small cycles we add to the basis as *local greedy cycles*. We present pseudocode for finding these cycles in Algorithm 1. We construct this cycle set by attempting to find a small cycle containing each edge using a truncated breadth-first search (BFS). Starting with all edges unmarked, the algorithm selects an unmarked edge and attempts to find a path between its endpoints. This search is truncated by bounding the number of edges searched so that each search is constant work and constructing the entire set is $O(m)$ work. If found, this path plus the edge forms a cycle, which is added to the new cycle set, and all edges used are marked. Tables 4.1-4.2 show the number of local greedy cycles found for all the test graphs when the truncated BFS was allowed to search 20 edges. Greedy cycles were found in all the graphs except for tube1, all of whose

vertices had such high degree that searching 20 edges was not enough to find a cycle.

Algorithm 1 Local Greedy Finder

```
function LOCAL-GREEDY(G)
  for e_{i,j} ∈ E do
    if e_{i,j} unmarked then
      p_{i,j} = Truncated-BFS(G \ (e_{i,j}), i, j, max_edges)
      Add p_{i,j} + e_{i,j} to cycle set
      Mark all edges in p_{i,j} + e_{i,j}
    end if
  end for
end function
```

Adding additional cycles to the cycle basis requires new probabilities with which to sample all the cycles. Since in the unweighted case, the stretch of a cycle is just its total length, it seems natural to update cycles proportional to their length.

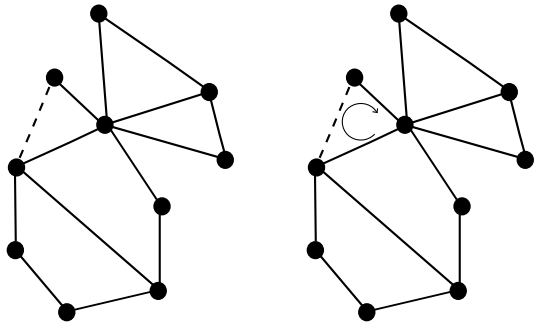


Figure 4.7: Local Greedy Cycles: An edge is selected on the left and a local greedy search is performed to find the cycle on the right.

4.3.3 Cycle Sampling and Updating in Parallel

In the original KOSZ algorithm, cycles are chosen one at a time with probability proportional to stretch. We propose a parallel update scheme in which multiple threads each select a cycle, at every iteration, with probability proportional to cycle length. Using unweighted graphs allows us to make this simplification as stretch is proportional to cycle length in the unweighted case. If two threads select cycles that share an edge,

one of the threads goes idle for that iteration. In Figure 4.8, threads 1, 2, and 4 select edge-disjoint cycles. However the third processor selects a cycle which contains edge 3, which is already in use by the cycle on thread 1. Processor 3 sits this iteration out while the other processors update their cycles.

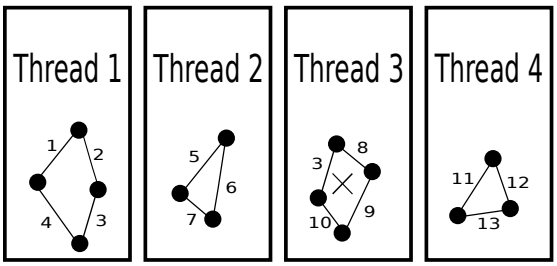


Figure 4.8: Example of Processors Selecting Cycles: Threads 1, 2, and 4 select edge-disjoint cycles, but thread 3 selects a cycle with edge 3 already in use. Thread 3 will go idle for an iteration.

We computed several measures of parallel performance. The first is simply the number of iterations. The second is the total work across all threads at every iteration. Lastly we report the span, or critical path length. This is the maximum of the work over all threads, summed over all the iterations.

We envision threads working in a shared memory environment on a graph that fits in memory. This might not be realistic in practice as there must be some communication of which edges have already been used which might be too expensive relative to the cost of a cycle update. However we are simply interested in measuring the potential parallelism, thus we ignore any communication cost.

The parallel selection scheme conditions the probabilities with which cycles are selected by the probability the cycle edges are available. So after choosing cycles with probability proportional to cycle length, and factoring in edge availability, the resulting cycle selection

probability is

$$p(C_e) = \frac{\text{length}(C_e)}{\sum_{C_i \in G} \text{length}(C_i)} p(e' \in C_e \text{available})$$

The probability that edges in a cycle are available is not calculated explicitly when the threads first select a cycle, but are there implicitly when a thread is forced to idle due to conflicting edges. This scheme creates a bias towards smaller cycles with less conflicting edges as more threads are added, which can increase total work.

4.4 Experiments with Non-Fundamental Cycle Sets

4.4.1 Experimental Design

We performed experiments on a variety of unweighted graphs from the UF Sparse Matrix Collection (the same set as in Section 4.2.1, shown in Tables 4.1-4.2). Again we distinguish between mesh-like graphs and irregular graphs. We also used a small test set for weak scaling experiments, consisting of 2D grids and BTER graphs.

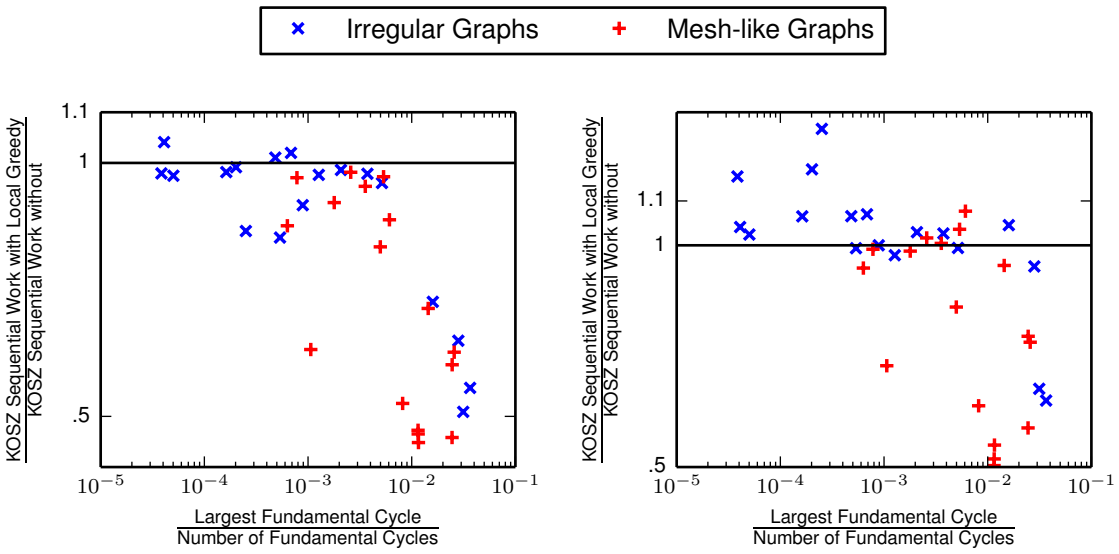
We continued to use our Python/Cython implementation of KOSZ, without a guaranteed low stretch spanning tree or a cycle update data structure. The code does not run in parallel, but we simulated parallelism on multiple threads by selecting and updating edge-disjoint cycles at every iteration as described above.

Our experiments consist of two sets of strong scaling experiments, the spanning tree cycles with and without local greedy cycles, up to 32 threads. We set a relative residual tolerance of 10^{-3}. Again we sacrifice accuracy to run more experiments on larger graphs. We consider the same four cycle update cost metrics as in Section 4.2.1: cycle length, $\log n$, log(cycle length), and unit cost updates. However in the case of the local greedy cycles, which cannot use the $\log n$ update data structure, we always just charge the number of edges in a cycle. For all the cost models, we measured the total work required

for convergence and the number of parallel steps taken to converge. For metric 4 these will be the same.

4.4.2 Experimental Results

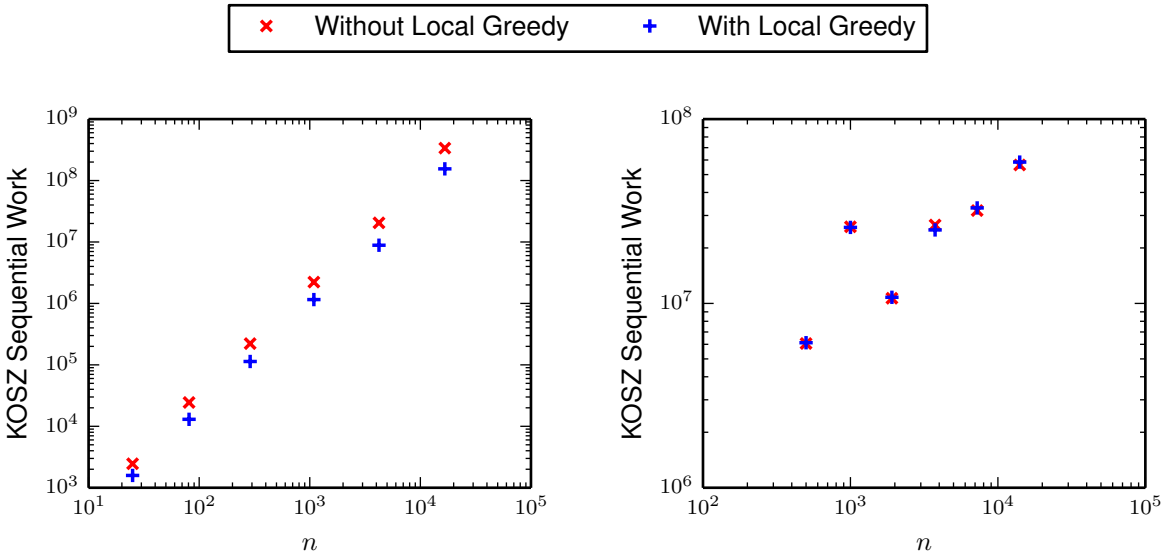
First, we examine the effects of using an expanded cycle set in the sequential algorithm. We estimate the usefulness of extra cycles as the length of the largest cycle in the fundamental set normalized by the number of cycles in the fundamental set. This is because we suspect the large cycles to be a barrier to performance, as they are updated the most frequently, and at the highest cost. Figure 4.9 shows the performance of the local greedy cycles for the two different graph types, using metrics 1 and 4. These plots show the ratio between the work of the expanded cycle sets as a function of the estimated usefulness. Points below the line indicate that adding local greedy cycles improved performance.



(a) Metric 1 (b) Metric 4

Figure 4.9: Sequential Comparison of Cycle Set Work: The ratio of KOSZ work with and without local greedy cycles, on one thread, is plotted against an estimate of the usefulness of extra cycles. Points below the line indicate that adding local greedy cycles helped.

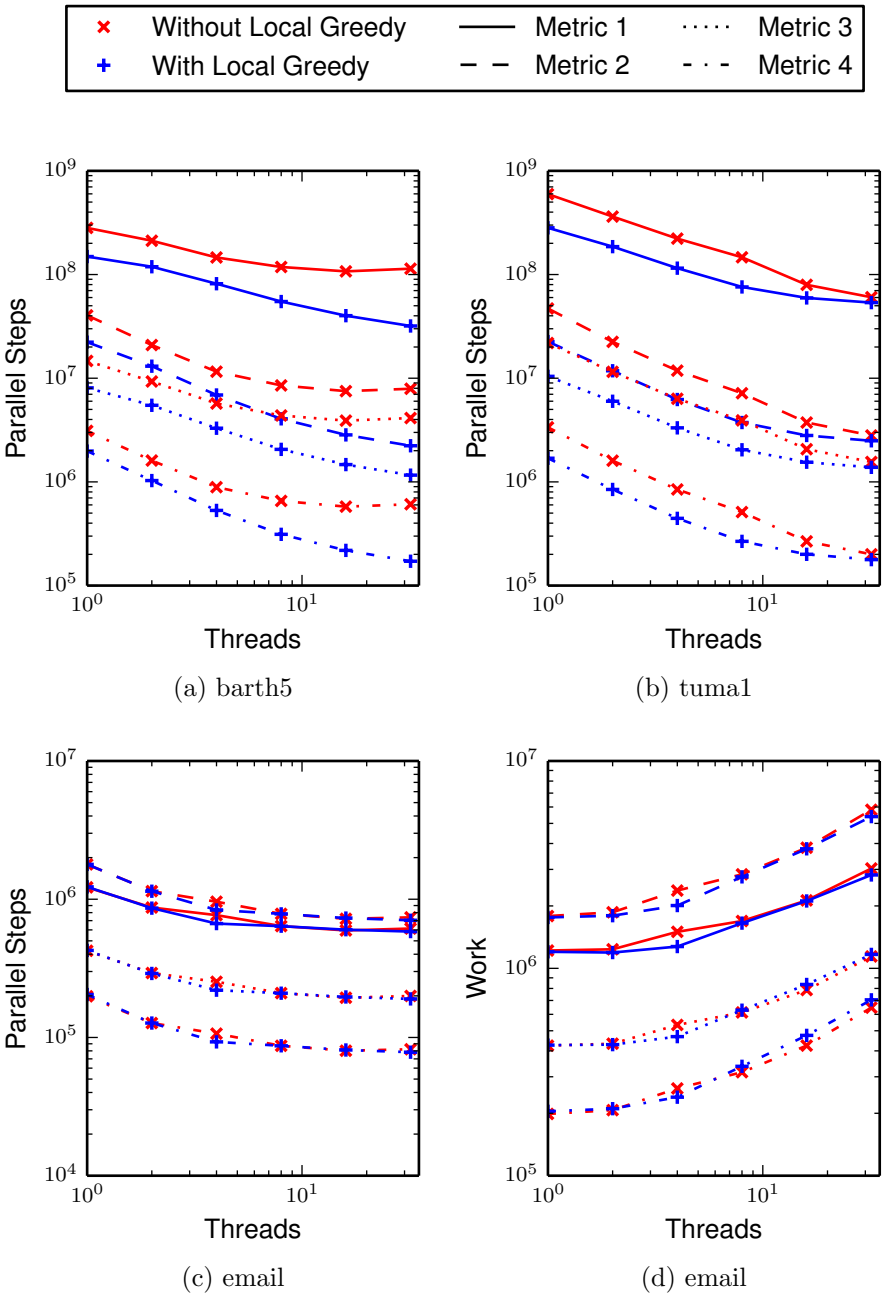
We examine how the local greedy cycles perform as graph size increases with weak scaling experiments on 2D grid graphs and BTER graphs. The 2D grids used for this experiment are the same as in Figure 4.6, and the BTER graphs were generated with the parameters: average degree of 20, maximum degree of $\sqrt{n}$, global clustering coefficient of 0.15, and maximum clustering coefficient of 0.15. Figure 4.10 shows the performance of cost metric 1 as graph size scales.



(a) 2D Grid (b) BTER

Figure 4.10: Weak Scaling of Cycle Set Work Under Cost Metric 1: The KOSZ work with and without local greedy cycles, on one thread, is plotted against the graph size in vertices.

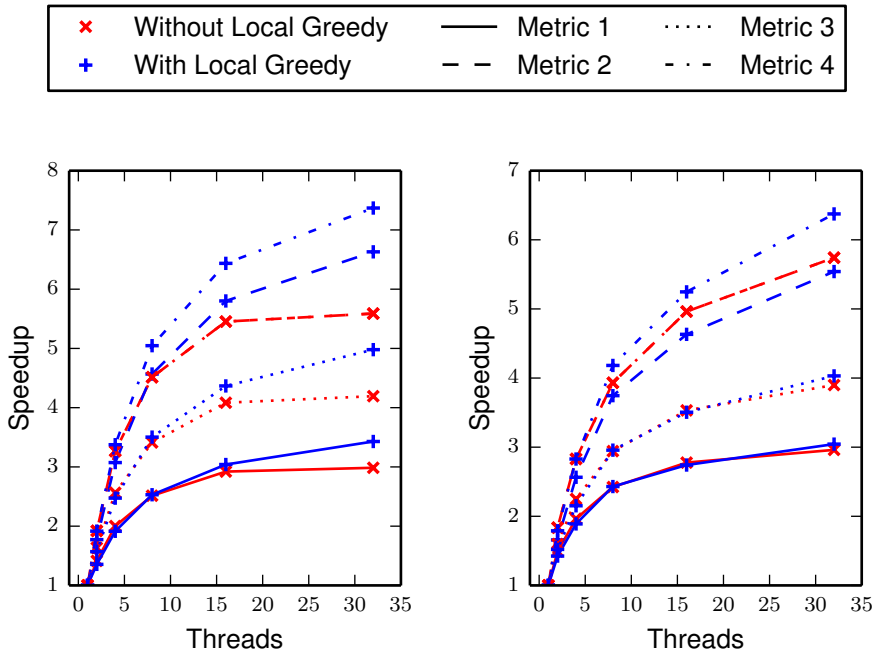
Figure 4.11 shows examples of our results on three of the graphs. In Figure 4.11(a) we plot the parallel steps (with the four different metrics) as a function of the number of threads used for the barth5 graph. The total edges (metric 1) is at the top of the plot, while the unit cost (metric 4) is at the bottom. These results are shown for both fundamental and extended cycle sets. Figures 4.11(b)-(c) show similar results for the tuma1 and email graphs. Figure 4.11(d) shows the effect adding threads has on the total work for the email graph.



(a) barth5 (b) tuma1

(c) email (d) email

Figure 4.11: Parallel Steps Scaling (shown for three example graphs) and Total Work Scaling of email Graph: As threads are added, parallel steps decreases for both cycle sets (steeper slope indicates better scaling). Unfortunately, as threads are added the total work increases for both cycle sets (ideally it would remain constant).



(a) Mesh-like Graphs (b) Irregular Graphs

Figure 4.12: Average Parallel Steps Speedup: The ratio of sequential work on one thread to parallel steps on multiple threads is plotted up to 32 threads.

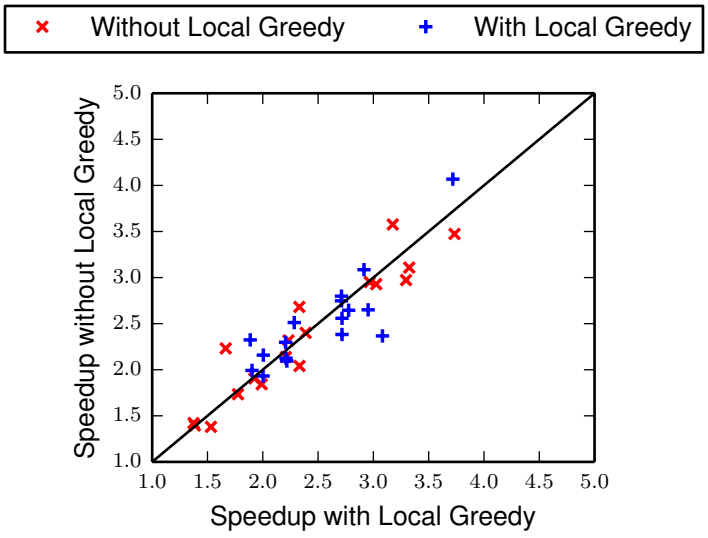


Figure 4.13: 8 Thread Speedup Comparison: The ratio of the 8 thread speedups of both cycle sets are plotted for all graphs (below the line local greedy speedup is better).

To measure the parallel performance across multiple graphs we look at the average speedup of the parallel steps across all graphs. Speedup is defined as the sequential work using one thread over the number of parallel steps using a number of multiple threads. Figure 4.12 shows the speedup with and without extended cycles. Note that without local greedy cycles metric 2 and metric 4 speedup are the same as the costs differ by $\log n$. We compare the speedup between the different cycle sets for the different graph types in Figure 4.13. We only show results for metric 1. We plot the speedup of using 8 threads without local greedy cycles against the speedup of using 8 threads with local greedy cycles.

4.4.3 Experimental Analysis

In the sequential results shown in Figure 4.9, there seems to be a threshold of largest cycle length above which local greedy cycles can be useful, but below which there is not much difference. However, there is not a clear scaling with the cycle length ratio, indicating that this was a crude guess as to where the extended cycle set is useful. Also note that mesh-like graphs tend to have larger girth (max cycle length) than irregular graphs, leading to local greedy cycles working better on meshes. The local greedy cycle improvement was slightly better for metric 1 where we count every edge update. At the other extreme, when updating large cycles is the same cost (unit) as small cycles added by local greedy, the local cycles were less effective. However there was still an improvement in number of cycles updated. We were unable to find some measure of the usefulness of a single local greedy cycle.

The weak scaling experiments shown in Figure 4.10 indicate that, with the exception of a BTER outlier, the work scaled nicely with graph size. Also results for both cycle sets scaled similarly. The extended cycle set benefits the 2D grid graphs while the BTER

graphs see little improvement or are worse. This is consistent with Figure 4.9 since the BTER graphs are irregular and the 2D grids are mesh-like.

The scaling of parallel steps plots show a variety of different behavior on the example graphs. On the mesh-like barth5 graph (shown in Figure 4.11(a)), the local greedy cycles improved both sequential performance and the scaling of parallel steps performance. At the left of this plot we see the extra cycles improved sequential results. Then as threads were added in parallel, the steeper slope indicates the local greedy cycles improved the scaling of the parallel steps. On the tuma1 graph (shown in Figure 4.11(b)), the local greedy cycles improved sequential performance, but resulted in similar or worse scaling. At the left of this plot we see the extra cycles improved results sequentially, but when scaled to 32 threads performance was similar. On the email graph (shown in Figure 4.11(c)), the local greedy cycles did not improve sequential performance, and scaling was poor with both cycle sets. There is little difference between the different cycle sets in this figure. Furthermore scaling was poor and quickly flattened out by about four threads. For a better understanding of the poor parallel steps scaling on the email graph, we examine the total work scaling (shown in Figure 4.11(d)), showing how much extra work we have to do when skewing the probability distribution. This extra work quickly increases, limiting the parallel performance.

In the average parallel steps speedup plot (shown in Figure 4.12), we see similar speedup for both cycle sets. On mesh-like graphs the local greedy cycles performed slightly better on all cost metrics beyond 16 threads. However on the irregular graphs, only with cycle cost metric 4 did the local greedy cycles perform better, and under metric 3 they performed worse. (Again note that without local greedy cycles metric 4 and metric 3 speedups are the same). We hypothesized that giving the solver smaller, extra cycles would improve the parallel performance compared to the fundamental cycles. However this seems to only be true for mesh-like graphs, and even then the improvement was

minimal. An interesting thing to note is that the speedup was better with the $\log n$ cost model. This is probably due to overcharging small cycles, which is less problematic when there are more threads to pick potentially larger cycles.

Taking a snapshot of the parallel steps speedup results on eight threads (shown in Figure 4.13), we see that there are some irregular graphs which did not have much speedup for either cycle set (bottom left of the plot). However there are mesh-like and irregular graphs which enjoyed a speedup for both cycle sets (top right of the plot). It is difficult to say on which graphs different cycles will aid parallelism.

4.5 Cycle Toggling Implementations

After our initial experimentation with KOSZ, we examine in more detail the underlying cycle update techniques, or cycle toggling. In the previous sections we considered four possible cost metrics for estimating cycle toggling work. Here we see how fast this can be done in practice. Fast cycle updates are essential to any practical implementation of KOSZ as it requires many cycle updates for energy minimization. In comparison PCG iterations are more computationally expensive but there are much fewer of them.

We need to efficiently support the following operations on a tree T, where each edge e is associated with a fixed resistance r_e and a flow f_e:

1. Query: Compute sums of r_e and $r_e f_e$ along a path in T.

2. Update: Increment all the flows on a path in T by Δ.

Although these updates are not adaptive, the result of each update does depend on all previous updates that interact with the path. This creates fundamental restrictions on cycle-toggling speed. This is especially true when considering any possible parallelism of updating multiple cycles simultaneously.

In the rest of this section we consider two different schemes for achieving fast cycle updates. The first uses data structures similar to the ones used by the KOSZ algorithm to update each cycle in $O(\log n)$ time. The second is a divide-and-conquer approach we introduce, which contracts the path based on preselected cycle updates.

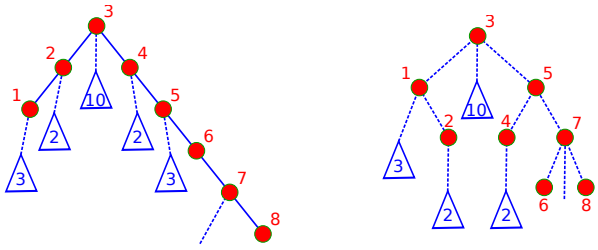
4.5.1 Reduction to Balanced BSTs

We elaborate on the data structure approach used by Kelner et al. [6], and give an overview of our data structure approach. The KOSZ data structures are based on top-down partitions of trees. Our implementations are based on a variant of this that uses binary search trees as building blocks. To help explain this, we first consider the easier case in which T is just a path, where we can solve the problem by building a static balanced binary search tree (BST) [74]. Any subtree in the BST corresponds to an interval in the path, which can be decomposed into a disjoint union of at most $2 \log n$ subtrees and nodes in the BST. To support our query and update operations, we add two pieces of information at every node v:

1. The *sum*, $\sum_i r_i f_i$ where $i \in$ the subtree containing v.
2. A lazy tag t, denoting the pending changes of flow in this subtree, caused by updates to parents.

The BST can answer the interval queries by adding up the *sum* fields of the corresponding subtrees. Note that this requires the *lazy tag* fields of all ancestors of the nodes added to be 0. This can be handled by 'pushing down' such fields as we access the BST. The updates involve modifying the *lazy tag* and *sum* fields of the subtrees correspondingly. This gives us a $O(\log n)$ per operation algorithm for the case where T is a path.

A classic way to generalize the path case to a tree is to use a heavy-light decomposition (HLD) [75]. Here, one first arbitrarily roots the tree. Then for every vertex u, we denote



(a) Heavy path re-rooted at separator vertex

(b) Virtual tree where heavy path is represented by the BST that stores it

Figure 4.14: One step of a heavy-light decomposition. Triangles are subtrees labeled with size.

v as the child of u whose subtree has the largest size (i.e. contains most vertices). We mark every edge (u, v) as *heavy* and say that all edges not marked *heavy* are *light*. An unextendable path of heavy edges is called a *heavy chain*. This decomposes the tree into heavy chains and light edges.

The key fact about this decomposition is that for any vertex v, its path to the root intersects at most $O(\log n)$ heavy chains and $O(\log n)$ light edges. Therefore, to support query and update operations on a tree, it suffices to handle the light edges and support these operations on heavy paths. For the latter, this is exactly the special path case and we can use BSTs described above. This leads to a theoretical time bound $O(\log^2 n)$ per operation, but a quite good running time experimentally.

This method is connected to the data structures used in KOSZ via virtual trees. Such a tree contains all the BST edges for heavy chains along with light edges. An example of creating a virtual tree from a HLD is shown in Figure 4.14. We can further optimize cycle updates by reducing the virtual tree height. A path between u and v in the original tree can be decomposed into the disjoint union of left-subtrees of nodes in the path between u and v in the virtual tree. In HLD, this virtual tree has height $O(\log^2 n)$ (since each

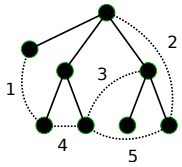
BST has height $O(\log n)$ and there are at most $O(\log n)$ heavy chains encountered in any path), so the time bound is $O(\log^2 n)$.

A better virtual tree can be constructed in a recursive manner. Consider the heavy chain starting from the root of T. Using the properties of heavy chains, one can prove that there exists a node v in the heavy chain, whose removal splits T into subtrees which have size at most half of the original tree size. We use v as the root of the BST for this heavy chain, and construct recursively. The virtual tree satisfies the property that any child has at most half the size of its parent, so it has height at most $\log n$. This gives us a $O(\log n)$ per operation algorithm. Compared to the recursive-separator based routine from [6], this scheme fixes the heavy path in addition to the root of the virtual tree. While this only changes the constants in the analysis, in terms of implementation it allows us to directly use the binary tree routine for paths mentioned above.

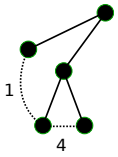
4.5.2 Recursive Divide-and-Conquer

The other main approach that we explore is a recursive divide-and-conquer scheme. The KOSZ algorithm treats cycle updates as an online process, a cycle is sampled, then updated, before another cycle is sampled. We consider the potential of an offline approach where we preselect N cycles, and use knowledge of this set to speed up the update of the set as a whole. This method recursively divides the N cycles in half until the subsets are each of size less than n. The cycles in the last level of the recursion are then updated in their preselected order.

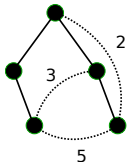
The speedup of this approach lies in the fact that we can reduce the problem to only the part of the graph involved in our preselected updates. We can further reduce the size of the graph by path contraction, condensing two edges if they are only updated when the other is updated. An example of this reduction and contraction is shown in



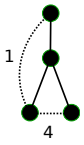
(a) Original Graph



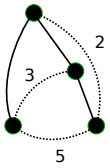
(b) Subgraph 1,4



(c) Subgraph 2,3,5



(d) Contraction of(b)



(e) Contraction of(c)

Figure 4.15: Illustration of graph reduction and contraction in divide-and-conquer. 5 cycles are preselected in the original graph(a) and divided into two groups, cycles (1,4) and (2,3,5). These cycles induce subgraphs (b,c) which only include edges and vertices of the relevant cycles. These subgraphs are then path contracted (d,e) to further reduce size.

Figure 4.15. This process results in several smaller graphs, where the cycles are updated, before pushing the cycle update information back up the recursive subgraph hierarchy. As this process resembles the recursive subgraph hierarchy of multigrid methods, we borrow the terms restriction and prolongation to describe the transfer of flow information up and down the hierarchy. This process is more formally captured in the following lemma.

Lemma 4.5.1 *Given a tree on n vertices, and N cycle updates, we can form a tree on $3N$ vertices, perform the corresponding cycle updates on them, and transfer the state back to the original graph. Furthermore, both the reduction and prolongation steps take $O(n)$ time.*

This procedure is identical to the greedy elimination, or partial Cholesky factorization steps from the ultra-sparsification routines [53]. Recursively dividing the cycle set yields

a recurrence of the form:

$$\mathcal{T}(N) = O(N) + 2\mathcal{T}(N/2),$$

which solves to $\mathcal{T}(N) = O(N \log N)$. If we set the size of our preselected cycle set to $O(n)$, then updating the entire set takes $O(n \log n)$ work, leading to a cost of $O(\log n)$ per update.

Unfortunately, the divide-and-conquer scheme does not parallelize naturally: the second recursive call still depends on the outcome of the first one. Furthermore, the bottleneck of this routine's performance is the restriction and prolongation steps, which unlike multigrid can not be reused when we resample another set. A large part of the expense is that vertices and edges must be relabeled as the graph is reduced. Doing this in random order leads to random access of vertex and edge labels. We try to optimize this by either compressing the memory of the graph storage, or by reordering the updates within each batch. In the case that the tree is just a path, much of the vertex and edge labeling can be done implicitly, reducing the overhead.

4.6 Experiments with Heavy Path Graphs

4.6.1 Heavy Path Graphs

Here we introduce a class of model problems that we will use to test and analyze different cycle-toggling approaches. These graphs are constructed by adding edges between vertices on a path graph. Edge resistances are selected so that the low-stretch spanning tree of the resulting graph is always the underlying path. As a consequence the edges on the path have larger edge weights than the off-path edges, so we refer to this class of graphs as *heavy path graphs*. An example of such a graph is shown in Figure 4.16.

Our interest in these problems does not come from any real world application. Instead

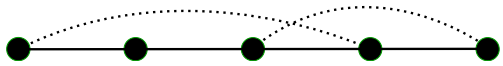


Figure 4.16: An example of a heavy path graph. The solid path edges are the low-stretch spanning tree of the graph.

we believe these are natural models to consider when studying KOSZ and other cycle-toggling algorithms. We believe that this model can be tuned to have various stretch properties along with spectral and graph separator properties, though we do not fully explore that in this dissertation. Furthermore they allow us to explore very fundamental questions about data structures and cycle-toggling implementations.

This model simplifies many of the implementation issues associated with dynamic trees, as the paths are easier to handle than more general tree layouts. Specifically, we can use a static, perfectly balanced binary tree for the path. This likely has the least data structure overhead as the optimum separator of an interval is implicitly the middle. Furthermore, this allows us to store the tree in heap order, which means the tree paths can be mapped to a subinterval using bit operations, and the downward/upward propagations can be performed iteratively.

Example Models

There are many possible subclasses that belong to the heavy path graph model. We introduce several subclasses here for experimentation.

1. **Fixed Cycle Length-1k**: These graphs are composed of a tree path with random resistances between 1 and 10,000, combined with off-tree edges between every pair $(i, i + 1000)$, e.g. an edge between vertices 1 and 1000, between vertices 2 and 1001, and so on.

2. **Fixed Cycle Length-2**: These graphs are composed of a tree path with random

resistances between 1 and 10,000, combined with off-tree edges between every pair $(i, i+2)$, e.g. an edge between vertices 1 and 3, between vertices 2 and 4, and so on.

3. **Random Cycle Length**: These graphs are composed of a tree path with random resistances between 1 and 1000, combined with n randomly selected off-tree edges, where n is the number of vertices.

4. **2D Mesh**: These graphs embed a tree path in a 2D mesh. The tree path resistances are chosen randomly between 1 and 1000.

5. **3D Mesh, Uniform Stretch**: These graphs are similar to (4) but with a 3D mesh.

We then consider two different ways of setting resistances on the off-tree edges on all of the models above.

1. **Uniform Stretch** Resistances of off-tree edges are chosen so that stretch is 1 for every cycle.

2. **Exponential Stretch** Resistances of off-tree edges are chosen so that cycles have stretch sampled from an exponential distribution.

4.6.2 Experimental Design

We now describe empirical evaluations of the cycle-toggling implementations from Section 4.5 on the class of graphs described in Section 4.6.1. As we only experimented with these path models, we used cycle-toggling methods that will only work on a path, but we also employed more general versions that will work on any graph.

The four cycle-toggling implementations are as follows:

1. BST-based data structure for general graphs

2. Path-only BST decomposition

3. Recursive divide-and-conquer for general graphs

4. Path-only recursive divide-and-conquer

Additionally we implemented a preconditioned conjugate gradient with diagonal scaling to compare against the cycle-toggling methods. We implemented all of these in C++ and also developed a Python/Cython implementation of the general recursive method. All algorithm implementations, graph generators, and test results for this work can be found at *https://github.com/sxu/cycleToggling*. We also experimented with Hoske et al.'s [72] implementation of cycle-toggling.

We used all of the generators described in Section 4.6.1 to create different heavy path graphs with varying total stretch. We used vertex sizes of 5×10^4, 10^5, 5×10^5, and 10^6. For the fixed cycle length generators, we set $hop = 1000$, and for the random cycle length generators, we set the number of off-tree edges to $2n$. To give an idea of the various stretch properties of these graphs, we list the total stretch for size 10^6 in Table 4.3.

	Uniform	Exponential
Fixed Length-1k	1.01e6	1.12e6
Fixed Length-2	2.00e6	1.04e7
Random Length	2.00e6	1.30e7
2D Mesh	2.00e6	1.08e7
3D Mesh	3.82e6	2.27e7

Table 4.3: Total stretch for all graph models of size 10^6. For each of the model problems in 4.6.1, this table shows the total stretch of cycles formed by adding edges to the underlying path. The models were generated with weights to create cycles with uniform stretch (all cycles with stretch 1), and exponential stretch(cycles with stretch chosen from an exponential distribution).

We also generated right hand side vectors b in two different ways to obtain both local and global behaviors.

1. Random: Randomly select x and form $b = \mathcal{L}x$,

2. (-1,1): Pick b to route 2 units of electrical flow from the left endpoint of the path to the right endpoint.

We performed these experiments on Mirasol, a shared memory machine at Georgia Tech, with 80 Intel(R) Xeon(R) E7-8870 processors at 2.40GHz. Problems were solved to a residual tolerance of 10^{-5}.

4.6.3 Experimental Results

We first examine the asymptotic behavior of the cycle-toggling methods on all the test graphs. Figure 4.17 shows the number of cycles required for convergence as a function of total stretch. This figure only involves solves using the (-1,1) right hand side as this was always a more difficult case.

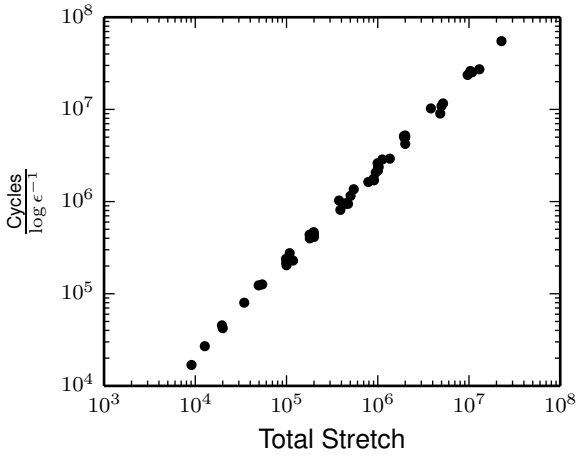


Figure 4.17: KOSZ asymptotic dependence on tree stretch. The number of toggles required by KOSZ is shown as a function of tree stretch. The reasonable slope indicates a lack of large hidden constants in KOSZ complexity.

We omit results from the Hoske et al. implementation because we found its performance to be slower by a factor of 50 than our cycle-toggling implementations. Their initialization costs are much higher than solve costs, making it prohibitively expensive to run on all of the test graphs in our set.

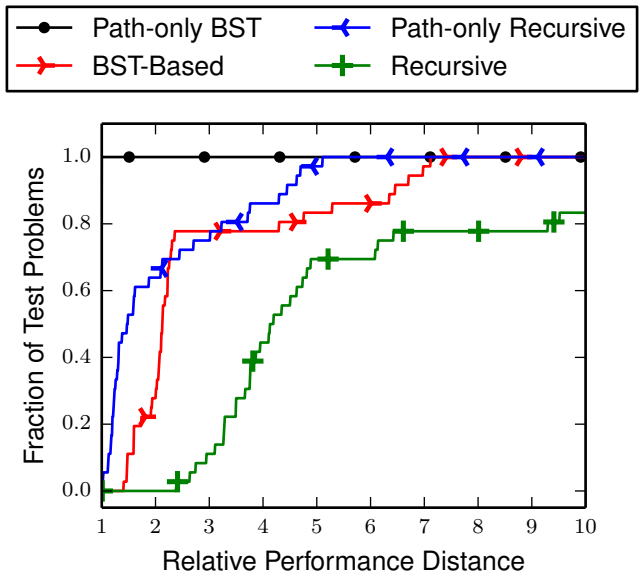


Figure 4.18: Performance profile of cycle-toggle time. The relative performance ratio of a method is its cycle-toggle time / best cycle toggle time for a single problem. This plot shows the fraction of problems that are within a distance from this relative performance ratio. The faster a method converges to 1 on this plot, the better its performance relative to the others.

To visualize the comparison of cycle-toggling implementations on all the different test graphs, we utilize a performance profile plot shown in Figure 4.18. We described performance profiles in more detail in Section 3.3.2, but to recap, a performance profile [68] calculates, for some performance metric, the relative performance ratio between each solver and the best solver on every problem instance. In our case the metric of interest is the average cycle-toggle time, so for each method and every graph, the relative performance ratio is the method's average cycle-toggle time divided by the lowest average cycle-toggle time over all methods. Then to capture how a method fares across the entire problem set, the performance profile shows the fraction of test problems (on the y-axis) that are within a distance (on the x-axis) from the relative performance ratio. This plot contains all the different model problems at every problem size tested.

Weak scaling experiments, measuring cycle-toggle performance as graph size increases, are useful for predicting performance on larger problems. The scaling behavior was

relatively similar across the model problems so we only show one example in Figure 4.19 for the 3D Unweighted Mesh with exponential stretch.

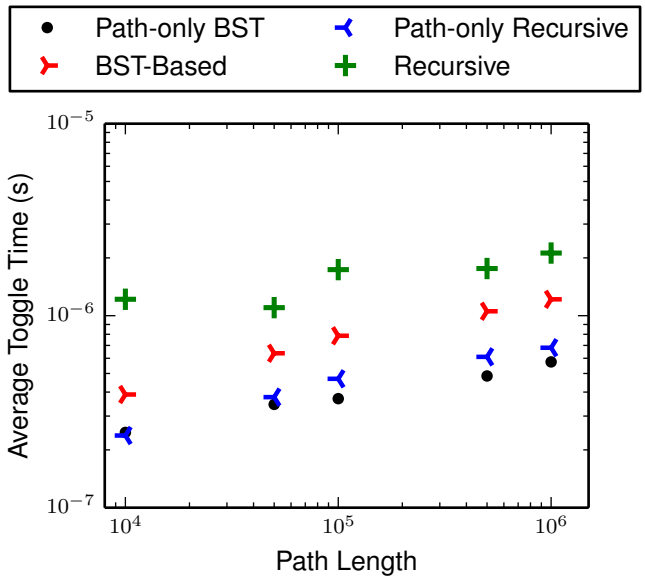


Figure 4.19: Weak scaling of cycle-toggle performance of all methods on unweighted 3D mesh model problems with exponential stretch. Average cycle-toggle time is shown as a function of problem size where an upward slope indicates decreased performance with larger problem size.

We examine how much time the recursive method spent restricting and prolonging flow in the recursive hierarchy, and how much time is spent doing cycle-toggles in Figure 4.20. Results are shown for the Fixed Length-1k model with a slightly wider range of problem size than the other experiments. The solve time in this plot includes the sum of the other operation timings, along with memory allocation. We performed this profiling with our Python/Cython implementation, but we believe the C++ performance is comparable.

Figure 4.21 shows BST-based cycle-toggle timing results relative to PCG results. Points below the line indicate cycle-toggling was faster, while points above the line are slower. This plot only includes size 10^6 problems using the (-1,1) right hand side. A random right hand side plot is omitted as these problems were much easier for both solvers, though slightly easier for PCG.

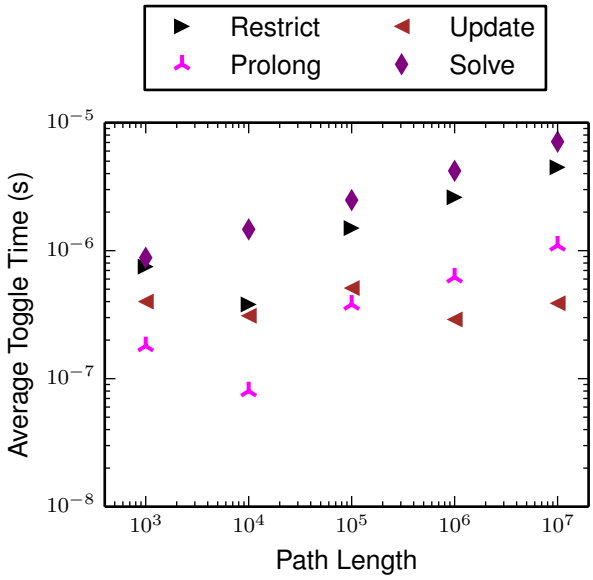


Figure 4.20: Weak scaling of cycle-toggle performance for the recursive solver on Fixed Length-1k model problems. Average cycle-toggle time is shown along with its most expensive sub-components: restriction, solve, and prolongation. Upward slopes indicate decreasing performance with problem size.

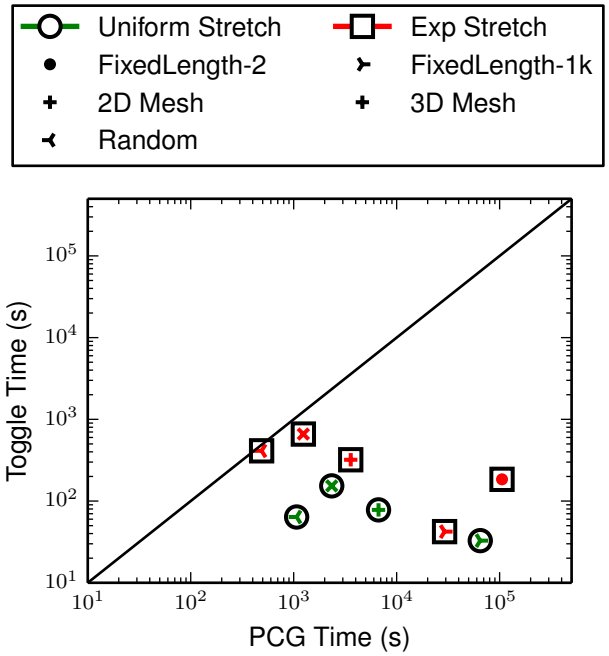


Figure 4.21: Comparison of BST-based data structure cycle-toggling to PCG by graph type. Points under the line indicate cycle-toggling method outperformed PCG.

4.6.4 Experimental Analysis

In Figure 4.17 the cycle-toggling methods' asymptotic dependence on tree stretch is near constant with a slope close to 1. Note that this plot would be linear even without the log axes. Concerning KOSZ practicality, it is highly important to see that there is not a large slope, which would indicate a large hidden constant in the KOSZ cost complexity. This plot tells us that with a combination of low-stretch trees and fast cycle update methods, dual space algorithms have potential. This figure also helps illustrate the range of problems we are using for these experiments. The stretch and resulting cycle cost both vary between four to five orders of magnitude.

The performance profile in Figure 4.18 indicates that the data structure based cycle-toggling methods performed the best using our implementations. For the path-only BST decomposition, the fraction of problems is already at 1 for a relative performance distance of 1, meaning that this was always the fastest. The path-only recursive method was slower, but still typically performed better than the general implementations, being half as fast as the path-only BST method on 60% of the problems. Comparing the two general implementations, the tree data structure is within a factor 4 of the best on 80% of the problems, whereas the recursive method is only within a factor of 4 on 40% of the problems. A distance of 10 indicates performance within the same order of magnitude, which the general recursive method achieved on 80% of the problems, indicating that these methods are competitive with one another.

The weak scaling experiments shown in Figure 4.19 do indicate a decrease in cycle-toggle performance as graph size increases. However, this plot is fairly optimistic, the largest performance decrease is about 2.5× as the graph size increases two orders of magnitude. The non steady plot for the general recursive solver suggests that the batch sizes were not scaled appropriately. Again, this plot is only for one of the graph models,

but most of them looked very similar to this.

Figure 4.20 helps identify the performance bottlenecks of the recursive method. The actual time spent updating cycles is less than the restriction and prolongation time. The restriction time is by far the most expensive, as it also includes time for relabeling edges and vertices. The scaling of this plot shows a stable update cost, with increasing restriction and prolongation costs. This method was designed to keep the update costs stable while increasing problem size, which seems to be case. Unfortunately the restriction and prolongation overhead costs are large and growing with problem size. Still, these operations are not highly optimized, and we wonder if we can borrow techniques from the multigrid community to speed them up.

The PCG experiments in Figure 4.21 indicate that KOSZ can outperform PCG on these heavy path models, using the (-1,1) right hand side. This class of problems had a wider performance gap for PCG than for the cycle-toggling routines, by about an order of magnitude. Furthermore, the graph property that causes difficulty for the solvers is different in each case; cycle-toggling has trouble on the graphs with exponential stretch, while PCG has difficulty with the fixed cycle length problems (Fixed Length-2 with uniform stretch even failed). These results suggest that heavy path graphs are a good direction to explore while searching for problems which could benefit from dual space optimization techniques like KOSZ.

4.6.5 Experimental Follow-up: Path Graph Exploration

After results in the previous subsection indicated the existence of heavy path graphs on which KOSZ outperforms PCG, we realized that these model problems could be of great interest to KOSZ solvers. Ideally we could design a heavy path graph in a way that we could independently control how difficult it is for either PCG or KOSZ. As PCG

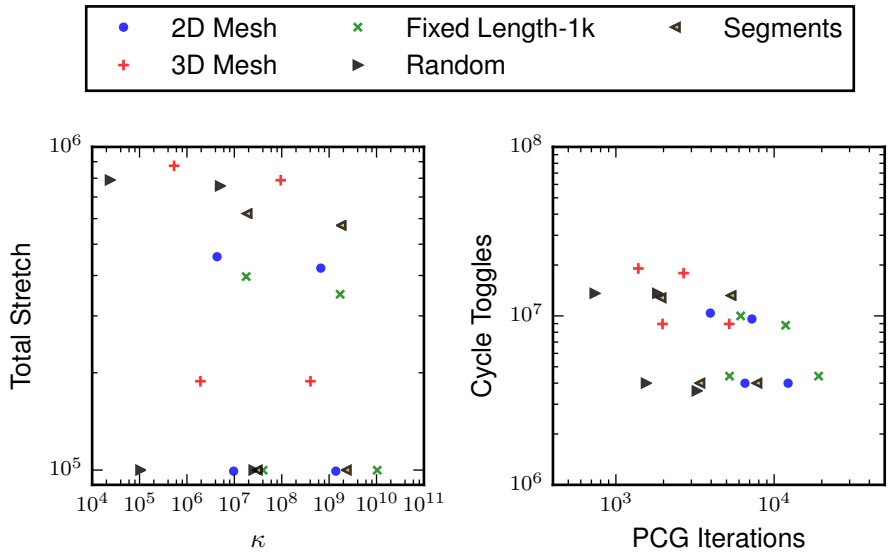
performance is bounded in terms of the matrix condition number, and KOSZ performance is dependent on stretch, our first intuition is to try and produce graphs where we can control both of these parameters. This section is mostly in the realm of future work, but we include some experimental results to give the reader some further intuition behind these models.

The models we described in Subsection 4.6.1 are already a good start towards this goal; separating the path graphs into those with uniform stretch and those with exponential stretch produce graphs problems that are easy and more difficult for KOSZ respectively. We still need a way to change the difficultly for PCG. The previous heavy path models all have random weights on the path edges, so we also add graphs with unweighted path edges. These tend to have lower condition number than weighted graphs. Thus for each of the heavy path graph topologies (embedded mesh, random cycle length, etc.) we have four edge weighting schemes.

1. Unweighted path edges + Uniform stretch off-path edges
2. Unweighted path edges + Exponential stretch off-path edges
3. Randomly weighted path edges + Uniform stretch off-path edges
4. Randomly weighted path edges + Exponential stretch off-path edges

While this does not give us a variable parameter for adjusting stretch and condition number, it does provide a binary parameter for each which is a good start. We show the effect of these parameters on the stretch and the condition number κ in Figure 4.22(a) for several of the model topologies on paths of length 10^5. One addition to the models previously described is the segments model, which divides the heavy path into several equal length segments, with no off-path edges going between segments. This was done to try and further worsen condition number. Each rectangular set of four markers in

Figure 4.22(a) is one of the four edge weighting schemes. Their relative location is always the same; the graphs with unweighted path edges are the two points to the left (weighted path edges are to the right), and the graphs with uniform stretch off-path edges are the two points on the bottom (exponential stretch off-path edges are on the top).



(a) Stretch and Condition Number (b) Resulting Performance

Figure 4.22: Controlling Path Graph Solver Behavior: Each set of four markers is a graph with the same edge topology, with different weighting schemes. The bottom-left point has unweighted path edges and uniform off-path resistance. The bottom-right point has weighted path edges and uniform off-path resistance. The top-left point has unweighted path edges and exponential off-path stretch. The top-right point has weighted path edges and exponential off-path stretch.

We experimentally verify that these parameters give us a difficulty switch for both PCG and KOSZ by showing the number of cycle toggles and the number of PCG iterations in Figure 4.22(b). We see that the rectangular arrangement of markers in Figure 4.22(a) roughly translates to Figure 4.22(b), confirming that the four edge weighting schemes can be used to change solver behavior somewhat predictably. Note that stretch is a much better measure of KOSZ difficulty than condition number is for PCG difficulty. Even if we instead look at the square root of the condition number of the preconditioned system, this is still only a worst case bound. We leave as an open question whether better heavy

path topologies or weighting schemes can provide better control over solver behavior.

4.7 Discussion

We implemented variations of the KOSZ algorithm to provide some of the earliest experimental results about this exciting theoretical development. While initial experiments indicated pessimism about the algorithm's potential, we considered modifications to improve performance in practice, such as using different cycle sets, or updating cycles in parallel. These modifications are promising on some test problems, but do not generalize as well as we would hope. The randomized nature of KOSZ makes it difficult to analyze and identify performance improvements.

We also examined different implementations of cycle toggling, the underlying primitive in the KOSZ algorithm. We compared sophisticated tree data structures against a divide-and-conquer approach and found that both methods are competitive, but data structures generally perform better. A batched divide-and-conquer approach could potentially be more useful if a single batch of cycles is reused, allowing the graph restriction and prolongation to be reused. This could be done if a batch of KOSZ cycle updates is used as a preconditioner inside another method for instance.

To test cycle toggling implementations, we proposed a set of model problems known as heavy path graphs. These simplify cycle toggling implementation details, but they also turn out to be interesting test cases where KOSZ can outperform PCG. We are hopeful that these might be arbitrarily tuned to show different solver behavior for KOSZ or PCG. Considering how to produce graphs with desired solver behavior partially inspired the genetic algorithm work in the next chapter.

Chapter 5

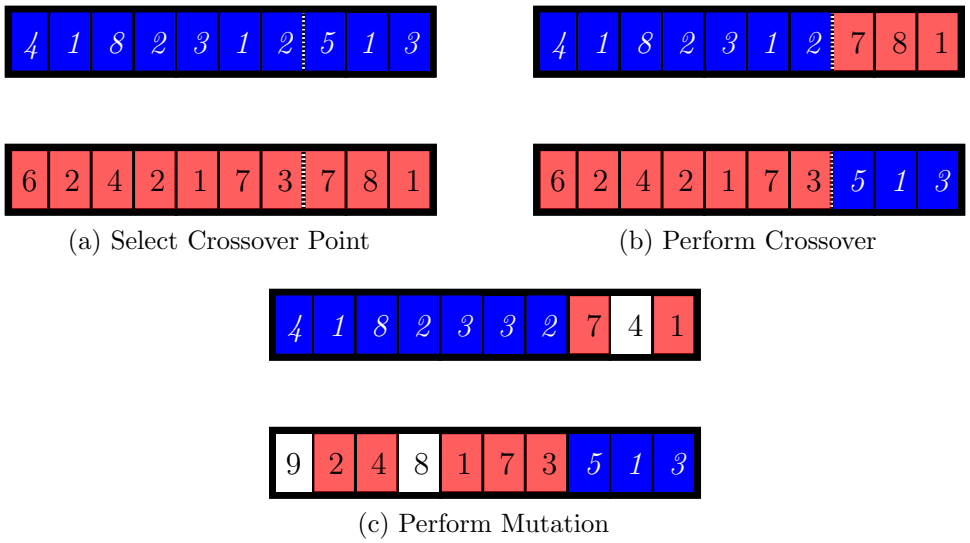
Searching for Difficult Test Problems

While the rest of this work focuses on the performance of Laplacian solvers on a fixed set of test problems, this chapter explores the test problems themselves. Here we seek to discover Laplacian linear systems that stress the ability of existing Laplacian solver packages to solve them efficiently. We employ a genetic algorithm, a tool from machine learning, to explore the problem space of graphs of fixed size and edge density. The goal is to measure the gap between theoretical and existing Laplacian solvers, by trying to find worst case example graphs for existing solvers. These problems may have little use inside any real world application, but they give great insight into solver behavior. We report performance results of our genetic algorithm, and explore the properties of the evolved graphs. This work can be found in our paper *Evolving Difficult Graphs for Laplacian Solvers* [76].

5.1 Genetic Algorithms

Genetic algorithms [77] are heuristic optimization techniques inspired by algorithms found in nature. These algorithms maintain a *population* containing possible solutions to a *fitness function*, the function the algorithm tries to optimize. A genetic algorithm evolves the population at each iteration by applying the following operators

- Selection: Select a subset of the population with the best fitness, those that maximize the fitness function evaluation. Members of this subset are known as *parents*, and will be allowed to contribute to the next generation.
- Crossover: Combine properties from pairs of parents to create children for the next generation.
- Mutation: Perform random changes on parents or their children to ensure genetic diversity in the next generation.



(a) Select Crossover Point

(b) Perform Crossover

(c) Perform Mutation

Figure 5.1: Crossover and Mutation Performed on Arrays: (a) Select the crossover point, which is the same for each array. (b) Crossover swaps the arrays at the crossover point. (c) Mutation randomly selects entries to modify, producing two child arrays.

The biological inspiration for genetic algorithms is the crossover and mutation that occurs during DNA replication. In the simplest description of a genetic algorithm, individuals in a population are arrays representing DNA sequences. Crossover splits two arrays at a fixed point, and swaps the sub-arrays. Mutation then changes some of the values in the arrays. Figure 5.1 illustrates array crossover and mutation. This idea can

be extended to algorithms with other underlying data structures. We will discuss the evolution of graphs below.

5.1.1 Our Genetic Algorithm

The goal of our genetic algorithm is to produce graphs that stress the performance of Laplacian solvers by requiring more work to solve. To design such an algorithm, we need to define crossover and mutation for graphs, and we need to define a fitness function tied to solver performance. While crossover and mutation are most easily defined for arrays, they can be defined for other structures. Our graphs only have information stored on edges, so we can perform mutations by randomly removing, adding, or changing edges, according to some mutation probability that we control. Performing crossover on graphs is less straightforward because unlike arrays, graphs generally cannot be separated into two pieces at a single point. Globus et al. designed genetic algorithms to produce graphs representing pharmaceutical drug molecules and simple digital circuits [78]. They addressed specifically the problem of performing graph crossover. Their solution was to separate graphs into fragments by finding random edge cut sets, and then reattach these fragments in a meaningful way. Their graphs were fairly small and these operations were tractable. We considered less expensive crossover operations, such as trading random or fixed sets of vertices or edges. We also considered omitting crossover altogether. For nontrivial crossover, we must also choose the crossover probability, the probability that two parents will produce a child through crossover.

We choose to perform crossover by simply trading an individual vertex and its incident edges between graphs, as this is the simplest nontrivial option. We also set the crossover probability to 1, in other words we allow all pairs of parents (graphs chosen by the selection operator) to create children. We also tested with no crossover and found very

poor evolution behavior. We plan to revisit graph crossover, but our current evolution operators seem to do a good job of exploring the search space. As for mutation, since we do not have a good intuition for the best mutation rate, we allow it to vary. We select a random number of edges between 1 and $n/10$ to remove, and randomly add enough back to keep the graph density constant.

Our genetic algorithm also requires a fitness function that will select graphs that are difficult for Laplacian linear solvers. At first this might be slightly counterintuitive since our fitness metric is to produce very unfit problems for another algorithm. However, in nature there might be pressure to evolve mice such that predators have a difficult time catching them. In this case, our predators are Laplacian solvers and our mice are graphs. The fitness function is more precisely the work required to solve a Laplacian linear system within a sufficiently small relative residual error $r = ||\mathcal{L}x - b||_2/||b||_2$. For all of the experiments in this paper we solve to a relative residual of 10^{-12}. It is also important to have a tiebreaker, so that if two systems require the same amount of work, the one with the larger residual will be ranked higher. We choose the right hand side b so that $b_1 = -1$, $b_n = 1$, and $b_i = -1 + 2 * i/(n-1)$. This was done to provide an ordering on the vertices so if we wanted to visualize output graphs there would be some consistency between graphs. Also this is typically a more difficult right hand side than a random vector because the initial residual error will tend to be higher. Experimentally, we find that using a different random right hand side every time can help avoid local maxima in the evolution. However, this is more difficult to analyze since it randomizes the fitness function.

We summarize our genetic algorithm as follows.

- Selection: Apply a Laplacian solver to all graphs in the population. Select as parents the problems which required the most work to solve to the target relative residual.

- Crossover: For every pair of parents, swap a randomly selected vertex to produce children.

- Mutation: Randomly select edges to remove, and replace them with new random edges. The number of edges chosen is also randomized between 1 and $n/10$. Replacement edge weights are randomized within an allowed range. Check if the graph is connected and only keep connected graphs.

A naive implementation would perform these three operators in order at every generation. This would require too much file I/O writing all the child graphs to disk before reading them back into memory to evaluate their fitness function. Instead we interleave the selection operator with the crossover and mutation operators by performing a Laplacian solve on each child graph as soon as it is produced in memory. Only graphs that are candidates for the next generation are written to disk.

5.2 Laplacian Solvers Used in This Study

The two solvers used in this study, PCG and KOSZ, are discussed in more detail in the rest of this dissertation, but we review them here. The conjugate gradient (CG) algorithm is a popular iterative method for solving the Laplacian linear system $\mathcal{L}x = b$ (and in general, any positive semidefinite linear system). It forms a sequence of improving approximations $\tilde{x}$ to x until convergence to within some tolerance $|\mathcal{L}\tilde{x}-b| < \epsilon$. Convergence is bounded in terms of the matrix condition number κ, which is the ratio of the largest to smallest nonzero eigenvalue. For Laplacian matrices the smallest eigenvalue is zero with multiplicity equal to the number of connected components of the graph. For connected graphs, $\kappa(\mathcal{L}) = \lambda_n(\mathcal{L})/\lambda_2(\mathcal{L})$. Typically a transformation called a *preconditioner* is applied to the linear system to improve the convergence behavior. The Jacobi preconditioner

simply scales the matrix symmetrically to make the diagonal elements all 1. Solving a Laplacian linear system with Jacobi PCG is equivalent to solving the normalized Laplacian with unpreconditioned CG. Therefore Jacobi PCG convergence is bounded by $\kappa(\mathcal{N}) = \lambda_n(\mathcal{N})/\lambda_2(\mathcal{N})$. We will examine how the genetic algorithm evolves the matrix spectra in Section 5.3.2.

KOSZ [6] was one of the first asymptotically fast Laplacian solvers to actually be implemented [69, 72] due to its relative simplicity. We skip the details here, but the algorithm randomly selects cycles from the graph and updates information on the edges of these cycles. These cycles are chosen with probability proportional to their *stretch* relative to a special tree known as a low-stretch spanning tree. The stretch of an edge is a measure of the penalty incurred traversing the spanning tree instead of the edge itself. The number of cycle updates required for KOSZ convergence is proportional to the quality of the tree, measured by its total stretch.

We will evaluate the performance of the solvers by the *work* metric, which roughly counts their arithmetic operations. This metric ignores memory access costs along with solver setup costs. The work required for Jacobi PCG is the cost of a matrix vector product plus the cost of diagonal scaling, multiplied by the number of iterations. For our experiments we use the Jacobi PCG inside the Ifpack2 and Belos packages of the Trilinos Project [64, 47, 48]. The work required for KOSZ is the sum of all the cycle update costs. KOSZ uses a sophisticated tree data structure to perform a single cycle update in $O(\log n)$ work on average. Instead of using this estimate, we measure the work inside the algorithm as the number of edges touched in the graph and the tree data structure during all of the cycle updates. For our experiments we use the KOSZ implementation by Haoran Xu for our previous study of KOSZ performance [70]. The low stretch spanning tree in this implementation is just a maximum-weight spanning tree, which is a good heuristic on the small graphs we generate here.

5.3 Genetic Algorithm Targeting Jacobi PCG

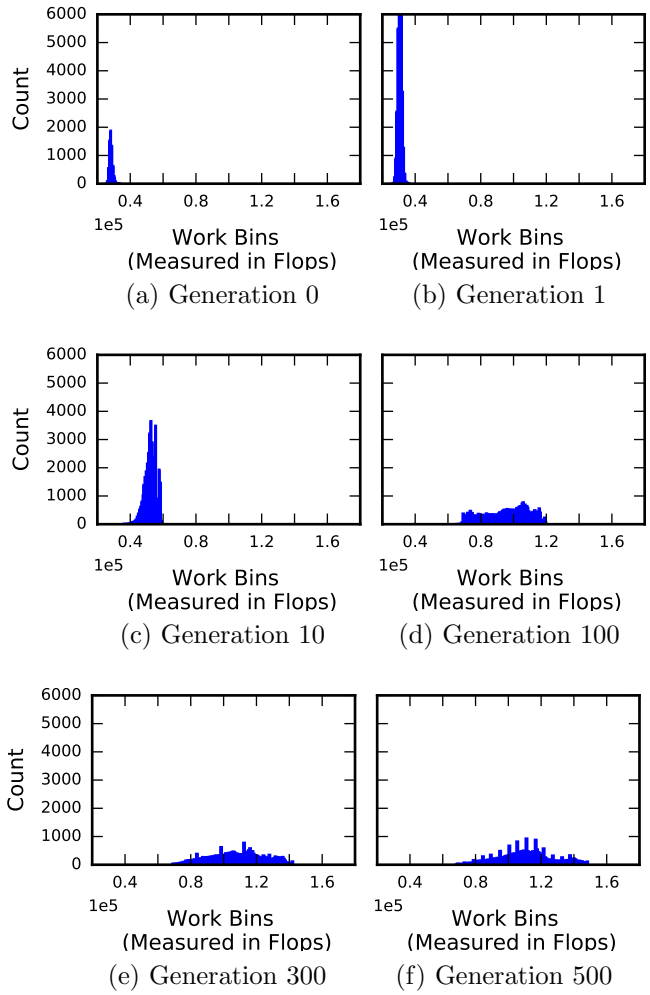
For the initial tests of our genetic algorithm we set the fitness function to maximize work required for Jacobi PCG. Initially we experimented with populations of graphs containing 100 vertices and 250 edges (600 nonzeros in the Laplacian matrix). We randomly generated around 30,000 connected graphs with edge weights chosen randomly between 1 and 1000. The selection phase chooses the worst 100 to be parents. Each pair of parents produces a child, which is mutated 5 times to produce children for the next generation. The process then repeats itself, starting with selecting another 100 parents. The evolution stagnated around generation 550, with very little gains after about generation 400, so we terminated the algorithm at generation 600. We divide the rest of this section into two parts. First we examine the behavior of the evolution, how the population changed over time and and how long this took. Then we examine the resulting graphs themselves.

5.3.1 Evolution Performance

To show how the population as a whole evolved from generation to generation, we include histograms of selected populations in Figure 5.2. The histogram binning is based on the amount of work required for convergence, so more difficult graphs are in bins towards the right. These plots indicate our algorithm is producing graphs which are more difficult for Jacobi PCG as the population evolves. The most difficult problems require about an order of magnitude more flops after at least 300 generations. We also see the histograms spread out as the evolution progresses, indicating that many of the mutated children will be much easier problems than graphs we are trying to find. Furthermore, the viable (connected) population shrinks over time. At generation 10, 36,000 children were produced, but by generation 100 there are only around 28,900 children. This could

indicate the algorithm is pushing graphs to be more disconnected.

Evolving ten generations required a little under four hours on a 2.3 GHz Intel Xeon processor. We have taken steps so the genetic algorithm runs in a reasonable amount of time, but more efficiency will be needed to scale this up to larger problems.



(a) Generation 0 (b) Generation 1

(c) Generation 10 (d) Generation 100

(e) Generation 300 (f) Generation 500

Figure 5.2: Evolution of Graphs that Challenge Jacobi Preconditioned Conjugate Gradient: Histograms show the performance, measured in work, of the graph population at select generations. The curve progresses to the right as problems become more difficult, eventually by an order of magnitude. The curve flattens out when most mutations are more likely to make the problem easier rather than harder.

5.3.2 Output Population

After seeing that we can evolve graphs that cause problems for Jacobi PCG, we examine these graphs. Random graphs in our starting population required about 45 iterations and the worst graph produced by this evolution required 212 iterations. We know that Jacobi PCG performance is directly related to the eigenvalue distribution of the normalized Laplacian, with well separated eigenvalues hindering convergence. Our intuition is that the genetic algorithm will work to spread out the eigenvalues, and on graphs with 100 vertices we can calculate the whole spectrum to verify this, as shown in Figure 5.3 for the worst graph in the 1st, 10th, 100th, and 600th generations. Indeed the eigenvalues of the normalized Laplacian become more spread out during the course of the evolution. It is mostly the tail of the distribution that shifts, small eigenvalues get smaller, with little change in the largest eigenvalues. It is not just the smallest eigenvalue getting smaller; many of the smaller eigenvalues become more well separated as the evolution continues.

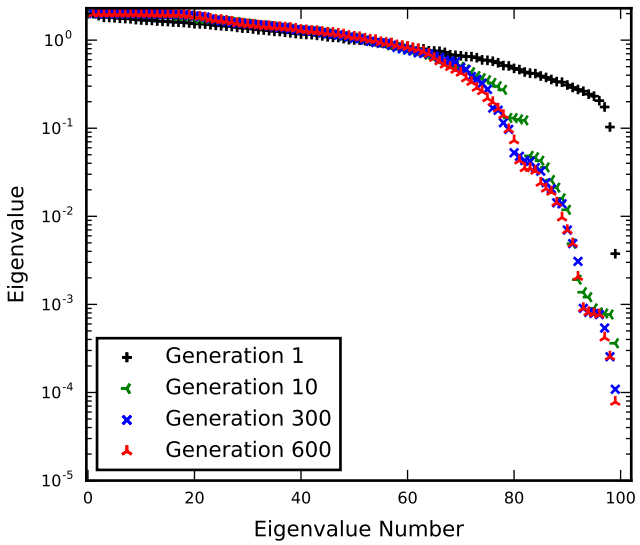


Figure 5.3: Normalized Laplacian Spectrum Evolution: The eigenvalues of the worst Jacobi PCG evolved matrices are shown at four generations in the evolution. As the eigenvalues spread out, convergence slows for Jacobi PCG.

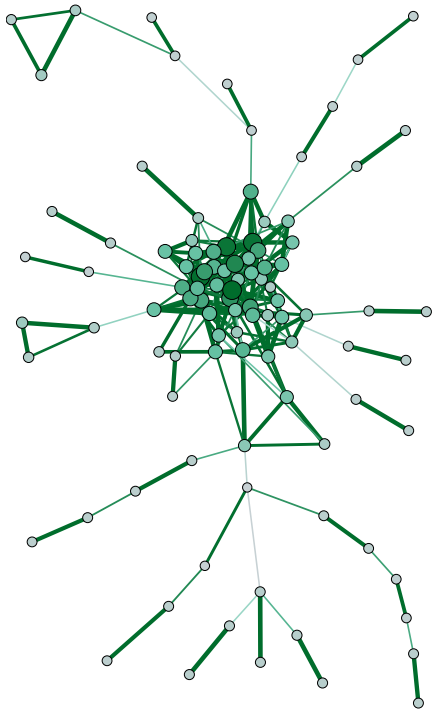


Figure 5.4: Visualization of Graph II: This is one of the worst (n=100, m=250) graphs for Jacobi PCG performance. Higher edge weights and weighted vertex degrees are represented by thicker and darker edges and vertices. Lower edge weights and weighted vertex degrees are represented by thinner and lighter edges and vertices.

To analyze graphs created from our evolutions, we use the NetworkX graph library [79]. Table 5.1 summarizes many of the statistics of these graphs. For clarity, we assign Roman numerals to the graphs in this table and refer to them by these labels throughout the paper. In Table 5.1, Graph I is a random graph used to initialize our genetic algorithm. Graph II is the worst graph produced during the evolution discussed in this section, aimed at maximizing Jacobi PCG work. We will discuss the other graphs later.

We first visualize Graph II to see if the structure can tell us anything interesting. For this task we use the Gephi graph visualization package for its plotting and analysis tools [80]. Using the Yifan Hu layout [81] provided by Gephi, we draw Graph II in Figure 5.4. We indicate higher edge weight and weighted vertex degree with thicker and darker edges and vertices. We indicate lower edge weight and weighted vertex degree with

thinner and lighter edges and vertices. Note that some of the edges are barely visible, but the graph is connected.

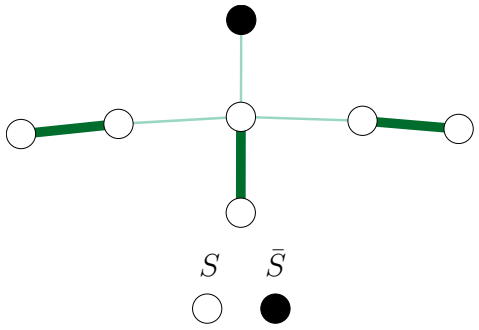


Figure 5.5: Graph II Minimum Conductance Set: The white vertices are the set S for which $\phi(S) = \phi(G)$. Low edge weights are shown with thinner and lighter lines; high edge weights are shown with thicker and darker lines. The low weight edge connecting S to $\bar{S}$, and the high weight edges between vertices in S result in a low $\phi(S)$

In Figure 5.4 we see that much of the low degree peripheral structure consists of alternating paths of low weight edges and high weight edges. We examine this phenomenon by considering the graph conductance $\phi(G)$, a useful property which estimates graph connectivity. The conductance of a set of edges S is a measure of the edge weights between S and the rest of the graph normalized by the volume of the set, more formally

$$\phi(S) = \frac{\sum\limits_{i \in S, j \in \bar{S}} w_{i,j}}{min\left(\sum\limits_{i \in S} d(i), \sum\limits_{i \in \bar{S}} d(i)\right)}.$$

The conductance of a graph is the minimum over all sets, $\phi(G) = min_S \phi(S)$. An important result in spectral graph theory is that the smallest nonzero eigenvalue of the normalized Laplacian $\lambda_2(\mathcal{N})$ is bounded above by the graph conductance, $2\phi(G) \geq \lambda_2(\mathcal{N})$. Our intuition is that the genetic algorithm will naturally modify conductance properties to minimize $\lambda_2(\mathcal{N})$. Indeed the conductance of Graph II is four orders of magnitude smaller than the random Graph I.

We draw the set $S \in$ Graph II that minimizes conductance and the edge that connects S to the rest of the graph in Figure 5.5. The edge between S and $\bar{S}$ is very small and there are very large edge weights between vertices in S, causing $\phi(S)$ to be small (and thus $\phi(G)$ to be small as well). Of course, there are small edge weights between vertices in S that could be increased to reduce $\phi(G)$ even further, but instead we see the alternating edge weight behavior common to the rest of the graph. While $\phi(G)$ is an upper bound on $\lambda_2(\mathcal{N})$, our hypothesis is that multiple low conductance sets throughout the graph will lower $\lambda_2(\mathcal{N})$ even further. This is why we see many paths with alternating high and low edge weights, which create many sets with low conductance. To test this idea, we created two new graphs, one with the larger path weights decreased to match the small ones, and one with the small weights increased to match the larger ones. In both cases the eigenvalues become more clustered, with an increase in the smallest nonzero eigenvalue. This is one example of how evolving graphs for specific solver behavior can give insight into graph structure. We will compare the structure of Graph II to other graphs in Section 5.5.

5.4 Performance Gaps Between Solvers

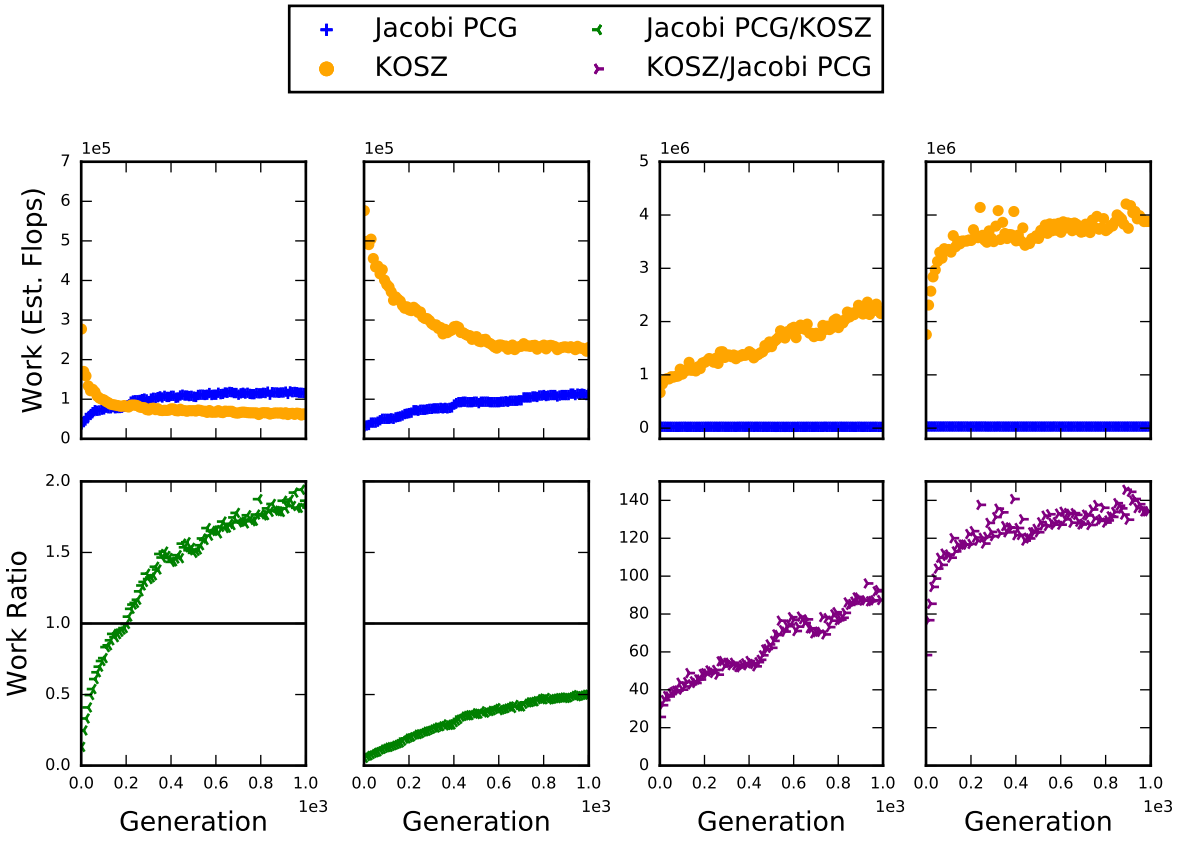
In our previous work we introduced a class of graphs for testing KOSZ called heavy path graphs, which simplified KOSZ implementation details [70]. KOSZ outperformed Jacobi PCG on several of these problems, which was an interesting discovery as previous experiments indicated pessimism regarding KOSZ performance [69, 72]. With this knowledge we asked on what other problems might KOSZ outperform traditional methods and by how much. In this section, we modify our genetic algorithm to explore solver performance gaps, and present the resulting evolution behavior. We defer an examination of the resulting graphs to the next section where we will compare them to other graphs.

A small change is needed from the genetic algorithm described in Section 5.1 to grow problems with large ratios between performance. We tweak the fitness function so that instead of selecting the problems that require the most work for a single solver, we select the problems with the largest ratio of work between two solvers. If the algorithm is trying to optimize the ratio of work between solvers A and B, it can increase the work of A, decrease the work of B, or both. We ran this modified algorithm on graphs of the same size as Section 5.3 (100 vertices, 250 edges) and also included a graph with the same number of vertices but higher density (100 vertices, 500 edges), both with edges with weights between 1 and 1000.

5.4.1 Evolution Results and Analysis

Figures 5.6(a) and (b) show the result of growing graphs with much worse Jacobi PCG performance relative to KOSZ performance. Figures 5.6(c) and (d) show the opposite, making KOSZ performance worse relative to Jacobi PCG. We choose to show the first 1000 generations of all these evolutions. By then they had either stagnated or were improving very slowly. Table 5.1 includes graph statistics for the less dense (m=250) graphs with maximized work ratios, labeled Graphs IV and V.

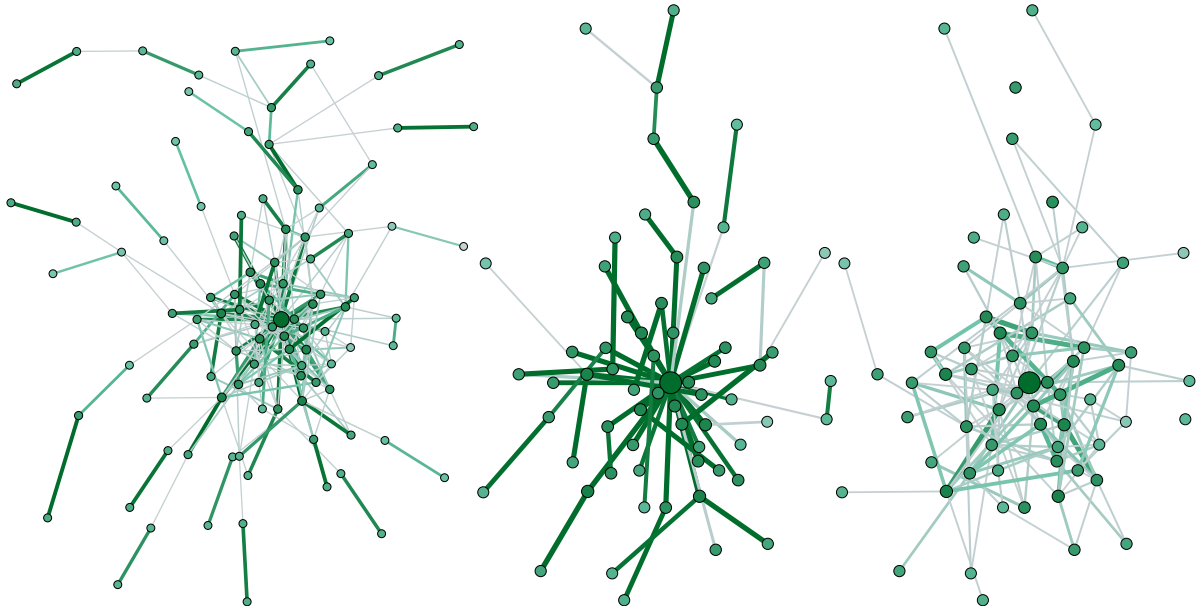
In Figures 5.6 (a) and (b), we see that the algorithm starts with graphs with a very small ratio between Jacobi PCG work and KOSZ work, indicating KOSZ performance is initially much worse. Then the algorithm successfully evolves graphs to increase the work ratio between solvers. Only on the less dense graphs, shown in Figure 5.6(a), does Jacobi PCG require more work than KOSZ, eventually by a factor of 2. Figure 5.6(b) indicates a similar trend in the more dense graph but stagnates at a point where KOSZ is still worse by a factor of 2. The evolution achieves these changes mostly by improving KOSZ performance, though the graphs become slightly harder for Jacobi PCG as well. In



(a) Jacobi/KOSZ (n=100, m=250)
(b) Jacobi/KOSZ (n=100, m=500)
(c) KOSZ/Jacobi (n=100, m=250)
(d) KOSZ/Jacobi (n=100, m=500)

Figure 5.6: Evolution of Graphs with Large Performance Ratios: Large ratios between Jacobi PCG and KOSZ are shown in (a) and (b). Large ratios between KOSZ and Jacobi PCG are shown in (c) and (d). The amount of work required for convergence is shown in the top plots. The work ratio is shown in the bottom plots, with a line at 1 indicating the same work required.

Figures 5.6(c) and (d), we see that the opposite evolution starts with the same work ratio, so we never expect KOSZ to outperform Jacobi PCG. Instead the algorithm produces graphs that are much worse for KOSZ, by as much as a factor of 140 for the denser graph in Figure 5.6(d). The algorithm makes little improvement to Jacobi PCG performance. Instead the large ratio is driven by very bad KOSZ performance.



(a) Full graph (b) 2-Core ∩ MST (c) 2-Core \ MST

Figure 5.7: Visualization of Graph IV: This graph has one of the largest performance gaps between Jacobi PCG and KOSZ; (a) shows the full graph, (b) shows the intersection of the 2-core with with maximum-weight spanning tree, (c) shows the 2-core with the MST removed (off-tree edges). Higher edge weights and weighted vertex degrees are represented by thicker and darker edges and vertices. Lower edge weights and weighted vertex degrees are represented by thinner and lighter edges and vertices.

5.5 Graphs and Their Effect on Solvers

Here we examine the graphs produced by the performance gap evolutions, and compare them with graphs produced by other evolutions. We refer the reader to Table 5.1 for all the graph statistics discussed throughout this section. Similar to Section 5.3.2, we will start with a visualization of Graph IV, the graph produced while maximizing Jacobi PCG work relative to KOSZ, which we draw in Figure 5.7(a). We seek to understand what makes this graph easier for KOSZ and more difficult for Jacobi PCG.

First, we examine what structural properties make Graph IV easier for KOSZ to solve. KOSZ work is directly proportional to the total tree stretch of the the spanning tree from which cycles are selected. To minimize stretch and improve KOSZ performance, edges

Label	I	II	III	IV	V
Fitness Function (Maximize Work)	None (Initial)	Jacobi PCG	KOSZ	$\frac{\text{JacobiPCG}}{\text{KOSZ}}$	$\frac{\text{KOSZ}}{\text{JacobiPCG}}$
Jacobi PCG Work	30.2K	149K	27.3K	122K	23.9K
KOSZ Work	256K	300K	2.06M	62.7K	2.22M
Average Weighted Degree	2,590	2,282	2,712	1,692	3,105
Clustering Coefficient	0.062	0.188	0.034	0.148	0.018
Triangles	21	114	15	140	7
Conductance (estimate)	1.0	0.00021	1.0	0.00061	1.0
Minimum Cut Weight	113.558	1.2212	11.1887	1.08926	53.3402
Total Stretch Relative to MST	523	321	723	61.9	761
2-Core Size	n=96 m=246	n=61 m=211	n=99 m=249	n=65 m=215	n=96 m=246
3-Core Size	n=83 m=220	n=50 m=195	n=89 m=229	n=53 m=192	n=92 m=238
4-Core Size	Empty	n=47 m=186	Empty	n=43 m=163	Empty
5-Core Size	Empty	n=41 m=163	Empty	n=31 m=117	Empty
6-Core Size	Empty	Empty	Empty	Empty	Empty
$\lambda_2(\mathcal{N})$	0.19051	0.00008	0.19277	0.00052	0.24699
$\lambda_n(\mathcal{N})$	1.81475	1.99993	1.81713	1.99949	1.79539
$\kappa(\mathcal{N}) = \frac{\lambda_n(\mathcal{N})}{\lambda_2(\mathcal{N})}$	9.52590	25561.0	9.42632	3882.28	7.26903

Table 5.1: Graph Statistics of Evolution Results: The most extremal graph (n=100, m=250) for each of the different fitness functions is shown here, along with a random graph in the initial population.

not belonging to the spanning tree should have much smaller weights relative to the tree edge weights. Indeed Graph IV has a much lower stretch than other graphs, about a factor of 8 less than the random Graph I. This can be seen in the graph visualization as well. We draw the tree edges and off-tree edges in Figures 5.7(b) and (c) respectively. We only show the vertices belonging to the 2-core as only the cycle structure is important for KOSZ. Indeed the tree edge weights are much larger than those of the off-tree edges, helping to explain the improved KOSZ performance. In the more dense case (n=100, m=500) we considered in Section 5.4, adding more edges creates more cycles which will naturally make the problems harder for KOSZ.

Next, we examine what makes Graph IV difficult for Jacobi PCG. We already asked this question of Graph II generated in Section 5.3.2, so we can start by looking at what these graphs have in common. As mentioned previously, well separated eigenvalues of the normalized Laplacian cause difficulty for Jacobi PCG. The matrix condition number k is a good estimate for eigenvalue separation and the normalized Laplacian of Graph II has a large k (from a very small $\lambda_2(\mathcal{N})$). The condition number of Graph IV is smaller than the condition number of Graph II by a factor of 7, however it is still 3 orders of magnitude larger than the condition number of random Graph I. The conductance is similarly low, and 4 orders of magnitude smaller than the conductance of the random Graph I. Figure 5.7(a) shows that Graph IV has similar alternating weight paths as Graph II, which could explain (see Section 5.3.2) Graph IV having many vertex sets of low conductance, helping to spread out the lowest eigenvalues. The graphs share several structural similarities such as having a large number of triangles, similar clustering coefficients, small cut weights, and similar core sizes up to the 5-core. Graph II does have a larger average weighted degree (total edge weight$*2/n$) which could help explain why it is more ill conditioned. Here we can see where the performance gap evolution used to generate Graph IV possibly had to make a trade off between lowering off-tree edge

weights for KOSZ, and producing large variations in edge weights to make the problem ill-conditioned for Jacobi PCG.

Lastly, we examine graphs that are difficult for KOSZ. Graph V was created in the performance gap experiments in the last section to maximize the ratio between KOSZ and Jacobi PCG performance. To round out the table, we generated Graph III to maximize KOSZ work, similar to the Jacobi PCG evolution in Section 5.1. Again, KOSZ performance is highly dependent on stretch, and these graphs both have higher stretch than the random Graph I by around a factor of 1.5. They have an order of magnitude higher stretch than Graph IV, which gives us a sense of the range of possible stretch values on graphs of this size, density, and weight range. One structural property contributing to higher stretch is the higher than random average weighted degree, indicating larger edge weights that can be used to form high stretch cycles. Another observation is that the 2-core of Graphs III and V are nearly the entire graph; there is no low degree peripheral structure similar to those found in Graphs II and IV. Almost all edges in the graph belong to a cycle, indicating longer cycles, which also contributes to higher total stretch. Note that the evolution hardly changes the spectral properties or conductance of Graphs II and IV from those of the random Graph I.

5.6 Discussion

This work demonstrates the use of a genetic algorithm to create graphs with desired properties. We target Laplacian solver behavior, but these ideas could be used to create graphs with other properties as well. Unfortunately, even on the relatively small graphs discussed here, these evolutions can be prohibitively slow if the fitness function evaluation is not trivial or if the population evolves slowly. To improve performance, we recommend re-evaluating crossover, mutation, and some of the other evolution parameters. We

also recognize the potential embarrassing parallelism of the fitness function evaluation. Hopefully with these changes, genetic algorithms will be feasible for larger graphs.

The output populations of graphs are of great interest to us in our ongoing study of Laplacian solvers. We successfully evolved a graph with five times more work for Jacobi PCG than a randomly generated graph of the same size and density. We also evolved a graph for which KOSZ outperformed Jacobi PCG by a factor of 2, adding to our limited knowledge of graphs where KOSZ could be useful. Further examination of these graphs gives insight into which structural graph properties are associated with extremal solver performance. Having many vertex sets with low graph conductance contributes to more spread out eigenvalues in the normalized Laplacian, slowing down Jacobi PCG convergence. KOSZ convergence is entirely dependent on the stretch of a low-stretch spanning tree. KOSZ performance improves when cycles are short and the off-tree edge weights are much lower than the tree edge weights. Some of these observations are not new; if we set out to design these graphs without a genetic algorithm we would know to include some of these features. However, our genetic algorithm can tell us how large the gaps between solver performance can be, and the exact combination of graph properties to achieve them.

There are several immediate directions to expand the study of these genetic algorithms. Of course more solvers could be used inside the fitness function, and some thought will have to go into how to generalize the work ratio fitness functions to more than two solvers. We suggest adding more graph statistics; on the small graphs discussed here, many global graph statistics are feasible to calculate. Also some thought should be given to the output graph population as a whole instead of just the worst graph. Do all of the graphs challenge Laplacian solvers for the same reasons, or is there more genetic diversity?

One of the initial visions of this work was to compile a collection of difficult graphs that could be glued together in different ways to create larger, and still relatively difficult

problems. For example a graph difficult for Jacobi PCG could be attached to a graph difficult for KOSZ to create a graph difficult for both solvers. This work is a first step towards assembling this collection.

Chapter 6

Discussion and Recommendations for Future Work

This dissertation presents experimental results on new and existing Laplacian solver algorithms and test problems in order to bridge the gap between theoretical and practical solvers. We conclude this work with a brief discussion of our findings and some suggestions for future research directions.

6.1 Laplacian Solver Experimentation

We have examined several existing Laplacian solver implementations and identified on which problems existing solvers are useful. We have also performed extensive experimentation with the KOSZ algorithm of Kelner et al., demonstrating interesting, albeit pessimistic results. As new solver implementations of more theoretical solvers become available, we hope the solver benchmarks reported here will be a useful start in comparing future performance. We have a few recommendations for future experimental work. We recommend consistency in how the Laplacian nullspace is handled, use of a more difficult right hand side than a random vector b, and an examination of convergence behavior in terms of the 2-norm residual, the $\mathcal{L}$-norm residual, and the error. We also recommend estimating solver work before spending effort optimizing wall clock time. We hope future

solver implementations are designed so that users have access to as much diagnostic information as possible.

6.2 Generating Difficult Test Problems

An important consideration of future Laplacian experiments is the design and inclusion of interesting or difficult test problems. While existing graph and matrix repositories have been incredibly useful for a variety of experimentation, we fear their overuse could limit our understanding of Laplacian algorithms. We have described two graph generation techniques for better understanding of Laplacian solvers, heavy path graph models and genetic graph evolutions. Both are interesting and deserving of future work. We believe that using a genetic algorithm to grow test problems is very promising for exploring the possibly exotic problem space where $\widetilde{O}(m)$ algorithms could be useful in practice.

6.3 Laplacian World Championships

One of the goals of our empirical solver work was to build the foundations for a larger, crowdsourced comparison effort, similar to a DIMACS challenge [82]. Sivan Toledo referred to this idea as the Laplacian World Championships [83], a competition where teams of theoretical computer scientists, applied mathematicians, and HPC experts would enter their solvers. These could be tested against a variety of difficult test problems and right hand sides. Different categories could exist to differentiate the best serial solver from the best parallel solver, or the best single use solver (setup time + per-solve time) from a solver used k times on the same problem matrix (setup time $+ k \times$ per-solve time). We still see this as a great way to bring theoretical algorithm developers together with high performance computing experts.

Bibliography

[1] E. G. Boman, D. Chen, B. Hendrickson, and S. Toledo, *Maximum-weight-basis preconditioners*, *Numerical Linear Algebra with Applications* **11** (2004), no. 8-9 695–721.

[2] K. D. Gremban, *Combinatorial preconditioners for sparse, symmetric, diagonally dominant linear systems*. PhD thesis, Carnegie Mellon University, 1996.

[3] E. G. Boman, B. Hendrickson, and S. Vavasis, *Solving elliptic finite element systems in near-linear time with support preconditioners*, *SIAM Journal on Numerical Analysis* **46** (2008), no. 6 3264–3284.

[4] N. K. Vishnoi, *Lx= b*, *Foundations and Trends® in Theoretical Computer Science* **8** (2013), no. 1–2 1–141.

[5] I. Koutis, G. L. Miller, and D. Tolliver, *Combinatorial preconditioners and multilevel solvers for problems in computer vision and image processing*, *Computer Vision and Image Understanding* **115** (2011), no. 12 1638–1646.

[6] J. A. Kelner, L. Orecchia, A. Sidford, and Z. A. Zhu, *A simple, combinatorial algorithm for solving SDD systems in nearly-linear time*, in *Proceedings of the 45th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 911–920, ACM, 2013.

[7] L. Katz, *A new status index derived from sociometric analysis*, *Psychometrika* **18** (1953), no. 1 39–43.

[8] C.-S. Liao, K. Lu, M. Baym, R. Singh, and B. Berger, *IsoRankN: spectral methods for global alignment of multiple protein networks*, *Bioinformatics* **25** (2009), no. 12 i253–i258.

[9] J. Leskovec, K. J. Lang, A. Dasgupta, and M. W. Mahoney, *Statistical properties of community structure in large social and information networks*, in *Proceedings of the 17th International Conference on World Wide Web (WWW)*, pp. 695–704, ACM, 2008.

[10] O. E. Livne and A. Brandt, *Lean algebraic multigrid (LAMG): Fast graph Laplacian linear solver*, *SIAM Journal on Scientific Computing* **34** (2012), no. 4 B499–B522.

[11] D. A. Spielman and S.-H. Teng, *Nearly-linear time algorithms for graph partitioning, graph sparsification, and solving linear systems*, in *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 81–90, ACM, 2004.

[12] I. Koutis, G. L. Miller, and R. Peng, *Approaching optimality for solving SDD linear systems*, in *Proceedings of the 51st Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 235–244, IEEE, 2010.

[13] R. Kyng and S. Sachdeva, *Approximate Gaussian elimination for Laplacians-fast, sparse, and simple*, in *Proceedings of the 57th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 573–582, IEEE, 2016.

[14] S.-H. Teng, *The Laplacian paradigm: Emerging algorithms for massive graphs*, in *International Conference on Theory and Applications of Models of Computation (TAMC)*, pp. 2–14, Springer, 2010.

[15] P. Christiano, J. A. Kelner, A. Madry, D. A. Spielman, and S.-H. Teng, *Electrical flows, Laplacian systems, and faster approximation of maximum flow in undirected graphs*, in *Proceedings of the 43rd Annual ACM Symposium on Theory of Computing (STOC)*, pp. 273–282, ACM, 2011.

[16] J. A. Kelner, G. L. Miller, and R. Peng, *Faster approximate multicommodity flow using quadratically coupled flows*, in *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 1–18, ACM, 2012.

[17] D. A. Spielman and N. Srivastava, *Graph sparsification by effective resistances*, *SIAM Journal on Computing* **40** (2011), no. 6 1913–1926.

[18] J. A. Kelner and A. Madry, *Faster generation of random spanning trees*, in *Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 13–21, IEEE, 2009.

[19] D. Zhou, J. Huang, and B. Schölkopf, *Learning from labeled and unlabeled data on a directed graph*, in *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, pp. 1036–1043, ACM/IMLS, 2005.

[20] N. L. D. Khoa and S. Chawla, *A scalable approach to spectral clustering with SDD solvers*, *Journal of Intelligent Information Systems* **44** (2015), no. 2 289–308.

[21] L. Orecchia, S. Sachdeva, and N. K. Vishnoi, *Approximating the exponential, the Lanczos method and an O(m)-time spectral algorithm for balanced separator*, in *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 1141–1160, ACM, 2012.

[22] S. Toledo, "The evolution of combinatorial solvers for Laplacian linear systems." SIAM Annual Meeting, 2014.

[23] Z. Feng, *Spectral graph sparsification in nearly-linear time leveraging efficient spectral perturbation analysis*, in *Proceedings of the 53rd Annual Design Automation Conference (DAC)*, pp. 1–6, ACM/EDAC/IEEE, 2016.

[24] R. Kyng, Y. T. Lee, R. Peng, S. Sachdeva, and D. A. Spielman, *Sparsified Cholesky and multigrid solvers for connection Laplacians*, in *Proceedings of the 48th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 842–850, ACM, 2016.

[25] F. R. Chung, *Spectral graph theory*, vol. 92. American Mathematical Society, 1997.

[26] D. A. Spielman, "Spectral graph theory." Course lecture notes currently available at http://www.cs.yale.edu/homes/spielman/561/, 2015.

[27] P. Arbenz, U. L. Hetmaniuk, R. B. Lehoucq, and R. S. Tuminaro, *A comparison of eigensolvers for large-scale 3D modal analysis using AMG-preconditioned iterative methods*, *International Journal for Numerical Methods in Engineering* **64** (2005), no. 2 204–236.

[28] A. Breuer, P. Gottschling, D. Gregor, and A. Lumsdaine, *Effecting parallel graph eigensolvers through library composition*, in *20th International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 466–473, IEEE, 2006.

[29] D. Kushnir, M. Galun, and A. Brandt, *Efficient multilevel eigensolvers with applications to data analysis tasks*, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **32** (2010), no. 8 1377–1391.

[30] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

[31] A. George and J. W. Liu, *Computer solution of large sparse positive definite systems*. Prentice-Hall, 1981.

[32] W. L. Briggs, V. E. Henson, and S. F. McCormick, *A multigrid tutorial*. SIAM, 2000.

[33] O. Axelsson and P. S. Vassilevski, *Algebraic multilevel preconditioning methods, II*, *SIAM Journal on Numerical Analysis* **27** (1990), no. 6 1569–1590.

[34] K. Watanabe, H. Igarashi, and T. Honma, *Comparison of geometric and algebraic multigrid methods in edge-based finite-element analysis*, *IEEE Transactions on Magnetics* **41** (2005), no. 5 1672–1675.

[35] K. Stüben, *A review of algebraic multigrid*, in *Partial Differential Equations*, pp. 281–309. Elsevier, 2001.

[36] A. Napov and Y. Notay, *An efficient multigrid method for graph Laplacian systems*, *Electronic Transactions on Numerical Analysis (ETNA)* **45** (2016) 201–218.

[37] A. Napov and Y. Notay, *An efficient multigrid method for graph Laplacian systems II: robust aggregation*, *SIAM Journal on Scientific Computing* **39** (2017), no. 5 S379–S403.

[38] P. M. Vaidya, "Solving linear equations with symmetric diagonally dominant matrices by constructing good preconditioners." manuscript, 1991.

[39] M. Bern, J. R. Gilbert, B. Hendrickson, N. Nguyen, and S. Toledo, *Support-graph preconditioners*, *SIAM Journal on Matrix Analysis and Applications* **27** (2006), no. 4 930–951.

[40] E. G. Boman and B. Hendrickson, *Support theory for preconditioning*, *SIAM Journal on Matrix Analysis and Applications* **25** (2003), no. 3 694–717.

[41] D. A. Spielman and J. Woo, *A note on preconditioning by low-stretch spanning trees*, *Computing Research Repository (CoRR)* (2009).

[42] I. Abraham and O. Neiman, *Using petal-decompositions to build a low stretch spanning tree*, in *Proceedings of the 44th Annual ACM Symposium on Theory of Computing (STOC)*, pp. 395–406, ACM, 2012.

[43] I. Abraham, Y. Bartal, and O. Neiman, *Nearly tight low stretch spanning trees*, in *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 781–790, IEEE, 2008.

[44] N. Alon, R. M. Karp, D. Peleg, and D. West, *A graph-theoretic game and its application to the k-server problem*, *SIAM Journal on Computing* **24** (1995), no. 1 78–100.

[45] M. Elkin, Y. Emek, D. A. Spielman, and S.-H. Teng, *Lower-stretch spanning trees*, *SIAM Journal on Computing* **38** (2008), no. 2 608–628.

[46] D. Chen and S. Toledo, *Vaidya's preconditioners: Implementation and experimental study.*, *Electronic Transactions on Numerical Analysis (ETNA)* **16** (2003) 30–49.

[47] M. A. Heroux, R. A. Bartlett, V. E. Howle, R. J. Hoekstra, J. J. Hu, T. G. Kolda, R. B. Lehoucq, K. R. Long, R. P. Pawlowski, E. T. Phipps, *et. al.*, *An overview of the trilinos project*, *ACM Transactions on Mathematical Software (TOMS)* **31** (2005), no. 3 397–423.

[48] M. A. Heroux and J. M. Willenbring, *A new overview of the Trilinos project*, *Scientific Programming* **20** (2012), no. 2 83–88.

[49] K. Deweese and E. G. Boman, *A comparison of preconditioners for solving linear systems arising from graph Laplacians*, *CSRI Summer Proceedings 2013* (2014) 3–9.

[50] E. G. Boman and K. Deweese, "A simple, parallel scheme for support graph preconditioning of networks." Poster presented at SIAM Parallel Processing for Scientific Computing (PPSC), 2014.

[51] Y. T. Lee and A. Sidford, *Efficient accelerated coordinate descent methods and faster algorithms for solving linear systems*, in *Proceedings of the 54th Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 147–156, IEEE, 2013.

[52] R. Peng and D. A. Spielman, *An efficient parallel solver for SDD linear systems*, in *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, pp. 333–342, ACM, 2014.

[53] D. A. Spielman and S.-H. Teng, *Nearly linear time algorithms for preconditioning and solving symmetric, diagonally dominant linear systems*, *SIAM Journal on Matrix Analysis and Applications* **35** (2014), no. 3 835–885.

[54] I. Koutis, G. L. Miller, and R. Peng, *A nearly-m log n time solver for SDD linear systems*, in *Proceedings of the 52nd Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 590–598, IEEE, 2011.

[55] S. Kaczmarz, *Angenäherte auflösung von systemen linearer gleichungen*, *Bulletin International de l'Academie Polonaise des Sciences et des Lettres* **35** (1937) 355–357.

[56] T. Strohmer and R. Vershynin, *A randomized Kaczmarz algorithm with exponential convergence*, *Journal of Fourier Analysis and Applications* **15** (2009), no. 2 262–278.

[57] S. Toledo, "An algebraic view of the new randomized Kaczmarz linear solver." Simons Institute Workshop, 2013.

[58] Y. Nesterov, *Efficiency of coordinate descent methods on huge-scale optimization problems*, *SIAM Journal on Optimization* **22** (2012), no. 2 341–362.

[59] E. G. Boman, K. Deweese, and J. R. Gilbert, *An empirical comparison of graph Laplacian solvers*, in *Proceedings of the 18th Workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 174–188, SIAM, 2016.

[60] T. A. Davis and Y. Hu, *The University of Florida sparse matrix collection*, *ACM Transactions on Mathematical Software (TOMS)* **38** (2011), no. 1 1:1–1:25.

[61] T. G. Kolda, A. Pinar, T. Plantenga, and C. Seshadhri, *A scalable generative graph model with community structure*, *SIAM Journal on Scientific Computing* **36** (2014), no. 5 C424–C452.

[62] C. Seshadhri, T. G. Kolda, and A. Pinar, *Community structure and scale-free collections of Erdős-Rényi graphs*, *Physical Review E* **85** (2012), no. 5 056109.

[63] P. F. Felzenszwalb and D. P. Huttenlocher, *Efficient graph-based image segmentation*, *International Journal of Computer Vision (IJCV)* **59** (2004), no. 2 167–181.

[64] E. Bavier, M. Hoemmen, S. Rajamanickam, and H. Thornquist, *Amesos2 and Belos: Direct and iterative solvers for large sparse linear systems*, *Scientific Programming* **20** (2012), no. 3 241–255.

[65] A. Prokopenko, J. J. Hu, T. A. Wiesner, C. M. Siefert, and R. S. Tuminaro, *MueLu users guide 1.0*, Tech. Rep. SAND2014-18874, Sandia National Labs, 2014.

[66] J. J. Hu, A. Prokopenko, C. M. Siefert, R. S. Tuminaro, and T. A. Wiesner, "MueLu multigrid framework." `http://trilinos.org/packages/muelu`, 2014.

[67] Y. Chen, T. A. Davis, W. W. Hager, and S. Rajamanickam, *Algorithm 887: Cholmod, supernodal sparse cholesky factorization and update/downdate*, *ACM Transactions on Mathematical Software (TOMS)* **35** (2008), no. 3 22.

[68] E. D. Dolan and J. J. Moré, *Benchmarking optimization software with performance profiles*, *Mathematical Programming* **91** (2002), no. 2 201–213.

[69] E. G. Boman, K. Deweese, and J. R. Gilbert, *Evaluating the dual randomized Kaczmarz Laplacian linear solver*, *Informatica* **40** (2016), no. 1 95–108.

[70] K. Deweese, J. R. Gilbert, G. L. Miller, R. Peng, H. R. Xu, and S. C. Xu, *An empirical study of cycle toggling based Laplacian solvers*, in *Proceedings of the 7th SIAM Workshop on Combinatorial Scientific Computing*, pp. 33–41, SIAM, 2016.

[71] R. Diestel, *Graph Theory*, vol. 173, pp. 23–28. Springer, 2012.

[72] D. Hoske, D. Lukarski, H. Meyerhenke, and M. Wegner, *Is nearly-linear the same in theory and practice? A case study with a combinatorial Laplacian solver*, in *Proceedings of the 14th International Symposium on Experimental Algorithms (SEA)*, pp. 205–218, Springer, 2015.

[73] P. A. Papp, *Low-stretch spanning trees*, undergraduate thesis, Eötvös Loránd University, 2014.

[74] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, pp. 779–784. MIT Press and McGraw-Hill, 2009.

[75] R. E. Tarjan and D. D. Sleator, *A data structure for dynamic trees*, *Journal of Computer and System Sciences (JCSS)* **26** (1983), no. 3 362–391.

[76] K. Deweese and J. R. Gilbert, *Evolving difficult graphs for Laplacian solvers*, in *Proceedings of the 8th SIAM Workshop on Combinatorial Scientific Computing*, SIAM, 2018. (to appear).

[77] M. Mitchell, *An Introduction to Genetic Algorithms*. MIT Press, 1996.

[78] A. Globus, S. Atsatt, J. Lawton, and T. Wipke, *Javagenes: Evolving graphs with crossover*, tech. rep., NASA, 2000.

[79] A. A. Hagberg, D. A. Schult, and P. J. Swart, *Exploring network structure, dynamics, and function using NetworkX*, in *Proceedings of the 7th Python in Science Conference*, pp. 11–15, 2008.

[80] M. Bastian, S. Heymann, and M. Jacomy, *Gephi: An open source software for exploring and manipulating networks*, in *Proceedings of the 3rd International AAAI Conference on Weblogs and Social Media (ICWSM)*, pp. 361–362, AAAI, 2009.

[81] Y. Hu, *Efficient, high-quality force-directed graph drawing*, *Mathematica Journal* **10.1** (2005) 37–71.

[82] D. S. Johnson, C. C. McGeoch, *et. al.*, *Network flows and matching: first DIMACS implementation challenge*, vol. 12. American Mathematical Society, 1993.

[83] S. Toledo, "The Laplacian world championships." Simons Institute Workshop, 2016.